

0 开发任务描述

- 设计并实现一个线下课程辅助系统。
- 大学中每位同学每学期会有多门不同的课程，课程分布在不同的教学楼甚至不同的校区；每门课程都会有一些课程资料、作业、考试和课程群等内容；此外每位同学在课外还会有一些个人或者集体的活动安排。
- 线下课程辅助系统可以帮助学生管理自己的课程和课外活动，具备课程导航功能、课程信息管理和查询功能，以及课外信息管理和查询功能等。每天晚上系统会提醒学生第二天上的课，每门课需要交的作业和需要带的资料，以及考试的信息；快要上课时系统根据该课程的上课地点设计一条最佳线路并输出；学生可以通过系统管理每门课的学习资料、作业和考试信息；在课外，学生可以管理自己的个人活动和集体活动信息，可以进行活动时间的冲突检测和闹钟提醒。

1 需求说明和分析

1.1 基本要求

- 校园内建筑物（教学楼、办公楼、宿舍楼）数不少于20个，其它服务设施不少于5种，共20个；
- 建立校园内部道路图
 - 包括各种建筑物、服务设施等信息；
 - 不能太简单（边数不能少于200条）；
 - 校区个数至少2个；
- 课程数目不少于10门
 - 每门课程包括上课时间、上课地点、课程教师、电子资料、纸质资料、作业信息和考试信息等；
- 课外活动不少于20个
 - 课外活动包括个人活动和集体活动，每个活动包括活动时间、活动地点等信息。

1.2 课程信息管理和查询

1.2.1 功能描述

- 课程基本信息包括：课程名称、上课时间、上课进度、上课地点、课程资料、已交作业、待交作业、课程群、考试时间和考试地点等信息
- 学生用户可以根据课程表查询课程的相关信息
- 学生用户可以上传、更新、下载作业内容。课程资料和作业支持查询、压缩和去重
- 管理员用户统一发布和管理课程（添加课程、修改课程）
- 管理员用户可以统一发布考试时间和考试地点

1.2.2 功能分析

- 根据上述功能需求，对于课程信息和文件资料需要查询和去重，可以考虑用平衡树实现高效的查询和去重。
- 对于文件资料的压缩，可以使用哈夫曼树实现文件压缩。
- 对于课程的基本信息，可以写一个 C++ 类来封装所有的基本信息，并给外部留好接口。

1.3 课外信息管理和查询

1.3.1 功能描述

- 课外活动包括个人活动和集体活动
- 支持学生用户输入添加课外活动
- 支持学生用户查询课外活动并排序
- 支持学生用户设置活动闹钟
- 支持检测活动冲突

1.3.2 功能分析

- 根据上述需求进行分析，我们可以写一个 C++ 类来封装课外活动的名称、类别、时间、地点等基本信息，并提供对外界的添加、查询、修改、删除的接口；
- 对于添加活动闹钟的需求，我们可以写一个单独的闹钟模块，将这个模块归入**模拟时间系统**，当我们增添活动时，调用闹钟模块的接口即可；
- 对于活动查询和检测冲突的需求，我们可以为所有的活动日程都建立一张日程表，这样就可以直观地看到所有的日程，方便查询和检测冲突（只要日程表上有重合部分就发生冲突）。这个日程表可以和课程系统的课程表共用，同时显示课程和活动；
- 对于检测活动冲突的需求，我们可以将其归纳为，判断时间区间是否出现重叠。对于这类区间修改、查询问题，我们可以用一棵线段树来维护所有的时间段，实现高效修改和查询；
- 对于查询结果进行排序的需求，我们可以用快速排序算法，对查询结果按照不同的关键字进行排序。

1.4 课程导航

1.4.1 功能描述

- 支持导入学校地图
- 支持学生根据课程或输入导航起止位置，自动规划导航路径
- 支持最短时间、最短路程、允许骑行等多种策略的寻路方式
- 支持跨校区最短路导航
- 支持途经多点的最短路导航

1.4.2 功能分析

- 对于学校的地图，我们可以从高德地图上获得沙河校区和西土城校区的具体道路、建筑数据，采用真实数据进行校区建模；对于跨校区的边，我们可以根据实际情况设置地铁（等间隔）、校车（定时）两种通行方式，和实际生活情况尽可能贴合，提高程序的实用性；
- 对于导航的基本功能，我们可以用 Dijkstra 算法进行求解，对于不同的导航策略，我们可将其视为计算边权的不同函数，每次通过传入不同的参数，选择不同策略；
- 对于跨校区边的最短路径，我们可以先在校区内部找到最短路，然后去搜索最近的、速度最快的跨校区边，搜索可以直接折半查找；
- 对于途经多点的最短路导航，我们先求出校区内任两点间的最短路，之后我们用动态规划的方法，求出最短路径；
- 对于多条最短路的情况，我们可以先通过 Dijkstra 算法获得最短路长度，之后进行深度优先搜索 DFS，按照已经求得的最短路进行剪枝，可以高效地求出所有最短路径。

1.5 模拟系统时间

1.5.1 功能描述

- 支持系统模拟时间推移，时间精度为小时，计算机10s对应系统模拟运行 1 h
- 支持时间加速和暂停

1.5.2 功能分析

- 由于模拟时间系统中不需要用到要求必须自行完成的数据结构以及算法，因此这部分我们完全可以放在前端借助qt便捷的库文件完成。
- 对于时间的推进，可以使用qt的qtimer库，该库提供了定时器，可以在设定的时间到后触发相关信号，因此我们只需要写好触发信号后的时间推进代码即可完成对时间的调整。
- 对于时间的加速和暂停，在加速时修改相关时间变量即可，对于暂停，则只需要暂停计时器。

1.6 日志文件

1.6.1 功能描述

- 记录用户所作操作和系统状态变化
- 记录输入输出信息
- 能够根据日志恢复现场

1.6.2 功能分析

- 考虑到本系统涉及到了模拟时间系统，因此要清晰的显示系统运行过程，理应同时记录现实时间以及模拟时间，基于以上观点，我们认为日志的格式应当是“现实时间 模拟时间 日志内容”并做持久化处理。

2 总体方案设计说明

2.1 软件开发环境

- 系统环境: Windows 10
- 开发语言: C++ 11
- 编译环境: g++ 4.9.2
- Qt 5版本: 5.0.2

2.2 软件结构

我们的软件主要分为：前端、后端两个部分，前端主要处理输入输出的图形化界面，后端主要处理整个项目软件的核心逻辑。

2.2.1 前端

- 前端采用Qt5框架进行编写
- 前端是与用户直接交互的、用户友好的图形化界面，主要方便用户使用，提供便捷的输入方式（按钮、选框、文本输入等），与图形化的视觉呈现界面。
- 前端负责账户权限的管理和更改，在学生账户下，可以查询通知、活动、课程、考试等信息；在管理员账户下，拥有发布通知、考试、添加修改课程信息、查看作业情况等权限。
- 前端还负责课程资料和作业的上传、压缩、下载操作，用户通过前端选择本地文件提交到系统，系统自动将提交的文件压缩保存；用户可以选择系统保存的文件下载，下载会自动解压缩。

2.2.2 后端

- 后端采用模块化的思想，将整个系统功能划分为四个模块。
- 后端采用 C++ 实现，仅使用了 C 标准库和 IO 输入输出流。具体实现了前端涉及到的算法和数据结构，并在此基础上完成了核心逻辑业务代码。后端本身就是一个功能完备（除文件上传下载）的系统。
- 后端分为五个模块：校园导航、日程安排、课程管理、时间模拟、日志文件。

2.3 模块划分

我们大致将整个任务划分为五个模块：校园导航、日程安排、课程管理、时间模拟、日志文件。

2.3.1 校园导航

实现校区内部、跨校区的导航，并输出路径。支持自定义起点终点、选择导航方式（最短时间、最短距离、是否骑车、途经多点等）。

2.3.2 日程安排

实现对所有课程和课外活动的时间管理，支持添加课外活动、判断时间冲突、删除课外活动；管理员添加考试时间和修改课程时间。

2.3.3 课程管理

实现课程的添加、删除（管理员）、信息查询、排序；实现对某一课程上传作业、下载资料、查询资料、作业查重、发布考试（管理员）。

2.3.4 时间模拟

时间系统实现时间加速、暂停、设置删除闹钟功能。

2.3.5 日志文件

日志系统记录程序运行的所有状态信息，能够根据日志文件恢复现场。

3 基础数据结构说明及分析

本部分包含了我们在项目中使用到的基础数据结构，包括线性表、结构体 Pair，串 String，变长数组 Vector 以及时间类 Time

3.1 线性表

3.1.1 一维数组 Array

说明：通过泛型编程实现的模板类，采用指针实现的一维定长数组，设置了3中不同的构造函数满足不同场景下的不同需求，同时对下标运算符和赋值运算符进行了重载，考虑到深拷贝以及自赋值问题，保证该类运行时稳定不出错。

成员函数声明如下：

```
template<class T>
class Array {
private:
    T *data;
    int maxN;
public:
    Array();
    explicit Array(int n);
    Array(int n, T x);
    ~Array();
    T& operator[](const int &index) const;
    Array<T>& operator = (Array other);
};
```

其中，所有待存储的数据项存储在成员变量 `T* data` 中，数量为 `maxN`。其接口函数包括：

(1) 构造函数

构造函数支持直接无参数构造数组、在已知数组长度的情况下构造数组、在已知数组长度和元素值的情况下构造数组。都是通过 `malloc` 函数申请内存空间实现的，具体实现参看代码部分：

```
template<class T>
Array<T> :: Array() {
    data = NULL;
}

template<class T>
Array<T> :: Array(int n) {
    data = reinterpret_cast<T*> (malloc(n * sizeof(T)));
    maxN = n;
}

template<class T>
Array<T> :: Array(int n, T x) {
    data = reinterpret_cast<T*> (malloc(n * sizeof(T)));
    maxN = n;
    for (int index = 0; index < n; index++)
        data[index] = x;
}
```

(2) 重载运算符 []

重载运算符允许数组被按照下标直接访问，和 C 语言自身的数组类似。代码实现如下：

```
template<class T>
T& Array<T> :: operator[](const int &index) const {
    return data[index];
}
```

(3) 重载运算符 =

重载运算符 = 可以比较两个数组是否相等，当且仅当它们长度相同且每个数组元素均相同时，才认为这两个数组相等，代码实现如下：

```
template<class T>
Array<T>& Array<T> :: operator = (const Array &other) {
    T *newData = reinterpret_cast<T*> (malloc(other.maxN * sizeof(T)));
    maxN = other.maxN;
    for (int index = 0; index < maxN; index++)
        newData[index] = other[index];
    data = newData;
    return *this;
}
```

3.1.2 二维数组 DyadicArray

说明：通过泛型编程实现的模板类，相较于一维数组增加了一个维度，是一维数组的扩展以满足更多的需求，其他方面与一维数组类似。

```
template<class T>
class DyadicArray {
private:
    T *data;
    int maxN, maxM;
public:
    DyadicArray();
    DyadicArray(int n, int m);
    DyadicArray(int n, int m, T x);
    ~DyadicArray();
    T* operator[](const int &index) const;
};
```

二维数组和一维数组没有太多差异，仅仅是申请空间时需要参数不同。略有差异的是重载运算符 []，它允许用户通过 DyadicArray[] [] 的形式访问数组元素，通过传入的下标参数和指针计算出对应数据项在内存中的位置，从而将对应数据项返回即可。代码实现如下：

```
template<class T>
T* DyadicArray<T> :: operator[](const int &index) const {
    return data + index * maxM;
}
```

3.1.3 测试用例及结果

对于一维数组与二维数组，我们以整型数据为例进行了测试，代码如下：

```
int main() {
    int n, m, k;
    cin >> n >> m >> k;
    Array<int> a(n, 0);
    for (int i = 0; i < n; i++)
        cin >> a[i];
    DyadicArray <int> c(m, k);
    for (int i = 0; i < m; i++)
        for (int j = 0; j < k; j++)
            cin >> c[i][j];
    Array<int> b = a;
    for (int i = 0; i < n; i++)
        cout << b[i] << " ";
    cout << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++)
            cout << c[i][j] << " ";
        cout << endl;
    }
    system("pause");
    return 0;
}
```

输入：

```
5 2 3
1 2 3 4 5
7 9 0
2 3 4
```

输出：

```
1 2 3 4 5
7 9 0
2 3 4
```

```
E:\Project\CourseAuxiliarySystem\src\test.exe
5 2 3
1 2 3 4 5
7 9 0
2 3 4
1 2 3 4 5
7 9 0
2 3 4
请按任意键继续...
```

3.2 结构体 Pair

说明：该类是对于C++ STL中pair类的仿写，同样采用泛型编程的方法，可以将两个数据统一使用、调用，同时完成了该类赋值以及部分比较运算符的重载。

```
template <class T1, class T2>
class Pair {
public:
    T1 first;
    T2 second;
    Pair() { }
    Pair(T1 k, T2 v): first(k), second(v) { }
    void operator = (const Pair<T1, T2> &other);
    bool operator < (const Pair<T1, T2> &other) const;
    bool operator > (const Pair<T1, T2> &other) const;
};
```

Pair 类主要需要注意的是重载了比较运算符，具体规则为：首先以 first 为第一关键字进行比较，若 first 相等，则以 second 为第二关键字比较。以 < 为例，代码实现如下：

```
template <class T1, class T2>
bool Pair<T1, T2> :: operator < (const Pair<T1, T2> &other) const {
    return first == other.first ? second < other.second : first < other.first;
}
```

测试略。

3.3 串 String

3.3.1 说明

该类同样是对于C++ String类的仿写，为操控字符串提供便利，其本质上是一个限定数据类型为char的链表并有着头尾指针，在类中，我们实现了赋值以及小于运算符的重载，重载+运算符为连接运算符，同时编写了成员函数用于类型转换，查看串长度等，进一步为操控字符串提供了便利。

```
int getSize(): 获取长度  
void pushBack(char c): 将字符插入串尾  
char* data(): 将String转换为char*
```

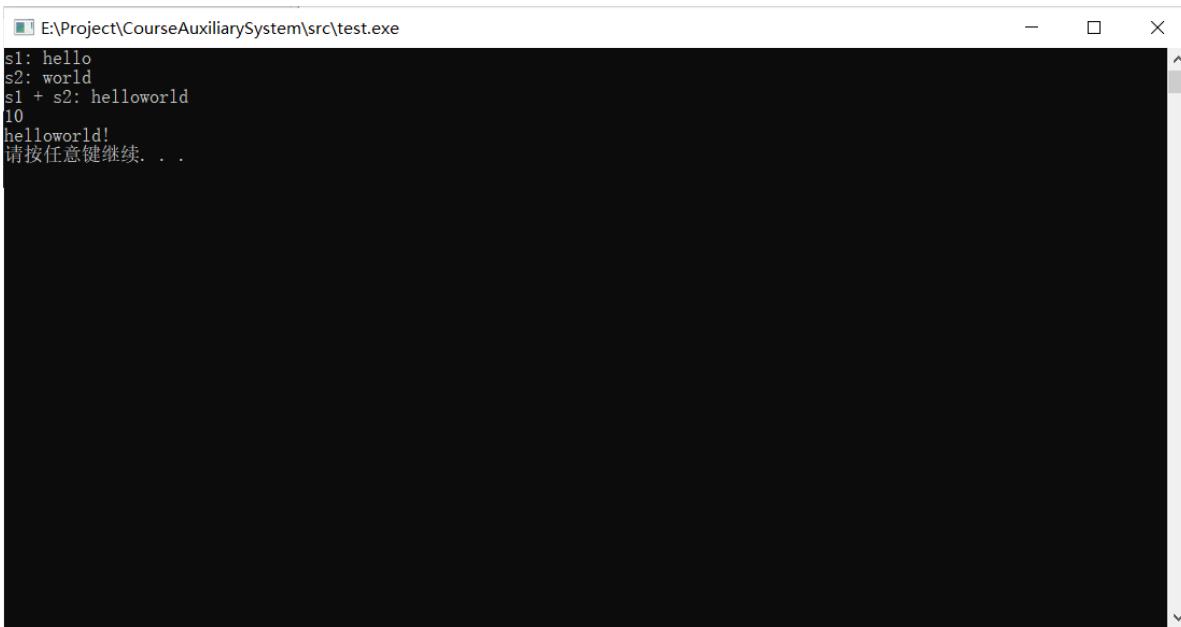
```
class String {  
private:  
    struct StringNode {  
        char c;  
        StringNode* next;  
        StringNode();  
        explicit StringNode(char x);  
        ~StringNode();  
    };  
    int size;  
    StringNode *head, *tail;  
public:  
    String();  
    explicit String(char* x);  
    ~String();  
    int getSize();  
    void pushBack(char c);  
    char* data();  
    String& operator = (String other);  
    String operator + (String other);  
    bool operator < (String other);  
};
```

3.3.2 测试数据及结果

测试代码如下：

```
int main() {  
    String s1, s2;  
    s1 = "hello";  
    s2 = "world";  
    cout << "s1: " << s1.data() << endl;  
    cout << "s2: " << s2.data() << endl;  
    String s3 = s1 + s2;  
    cout << "s1 + s2: " << s3.data() << endl;  
    cout << s3.getSize() << endl;  
    s3.pushBack('!');  
    cout << s3.data() << endl;  
    system("pause");  
    return 0;  
}
```

运行结果如下：



```
E:\Project\CourseAuxiliarySystem\src\test.exe
s1: hello
s2: world
s1 + s2: helloworld
10
helloworld!
请按任意键继续. . .
```

3.4 变长数组 Vector

3.4.1 说明

该类是对C++ STL中的Vector的仿写，是可以变长的数组，通过在设置不同的阈值，在超过某阶段阈值后重新开辟空间并将原有数据进行拷贝。同时提供数组反转，判断空数组等成员方法

```
template <class T>
class Vector {
private:
    T *data;
    int maxLength;
    int size;
    //重新分配数组空间
    void reNew(int newLength);

public:
    Vector();
    explicit Vector(int n);
    Vector(int n, T x);
    ~Vector();
    //获取数组元素数目
    int getSize();
    //添加新元素至数组尾部
    void pushBack(T e);
    //将数组末尾元素弹出
    void popBack();
    //判断数组是否为空
    bool isEmpty();
    T &operator[](int index);
    Vector<T>& operator = (Vector<T> other);
    Vector<T> operator + (Vector<T> other);
    //将数组逆序
    void reverse();
};
```

Vector 类主要在不能事先确定数组长度时，提供一种可变长数组的选项，通过核心的 reNew 函数重新分配空间，避免了数组长度过小导致数组越界，或数组长度过长导致占用过多内存资源的问题。其关键函数如下：

(1) 更新长度 reNew

reNew 函数负责在原本分配的空间不足时，重新为数组动态申请内存空间。具体策略为：首先默认申请大小为 64 的数组空间；如果目前分配的空间全部用完，还有新的元素加入时，则重新申请一个长度为原本两倍的空间，将原本的数据项全部拷贝过去，释放原本的内存空间。具体代码实现如下：

```
template<class T>
void Vector<T>::reNew(int newLength) {
    T *newData = reinterpret_cast<T *>(malloc(newLength * sizeof(T)));
    memset(newData, 0, newLength * sizeof(T));
    for (int i = 0; i < size; i++) newData[i] = data[i];
    free(data);
    data = newData;
    maxLength = newLength;
}
```

(2) 尾部插入函数 pushBack 和删除函数 popBack

pushBack 函数是尾部插入函数，首先会判断当前申请的内存空间是否已经被占满了，如果被占满了，则使用 reNew 函数更新当前的内存空间，再将元素插入数组的尾部；popBack 函数直接在尾部删除即可。代码实现如下：

```
template<class T>
void Vector<T>::pushBack(T e) {
    if (size == maxLength) reNew(maxLength << 1);
    data[size++] = e;
}

template<class T>
void Vector<T>::popBack() {
    if (size == 0) return;
    size--;
}
```

(3) 重载运算符 +

重载了+运算符，使得可以直接将一个 Vector 接在另一个 Vector 之后得到一个新的 Vector。具体实现是，将另一个 Vector 的每个元素依次 pushBack 到 Vector 中，代码实现如下：

```
template<class T>
Vector<T> Vector<T>::operator+(Vector<T> other) {
    Vector<T> ans = *this;
    for (int index = 0; index < other.getSize(); index++)
        ans.pushBack(other[index]);
    return ans;
}
```

(4) 其他接口函数

reverse 函数可以将 Vector 内的元素反转，具体而言，将 Vector 前后的元素依次使用 swapElement 函数交换即可。

getSize 函数可以返回 Vector 内的元素个数。

isEmpty 函数可以判断 Vector 是否为空，若为空，则返回1。

重载运算符 [] 可以用访问数组的方式访问 Vector 元素。

重载运算符 = 可以判断两个 Vector 是否相等，具体方法为，若 Vector 的每个元素都相等，则这两个 Vector 相等。

具体代码实现较为简单，可以直接参看源程序部分。

3.4.2 测试数据及结果

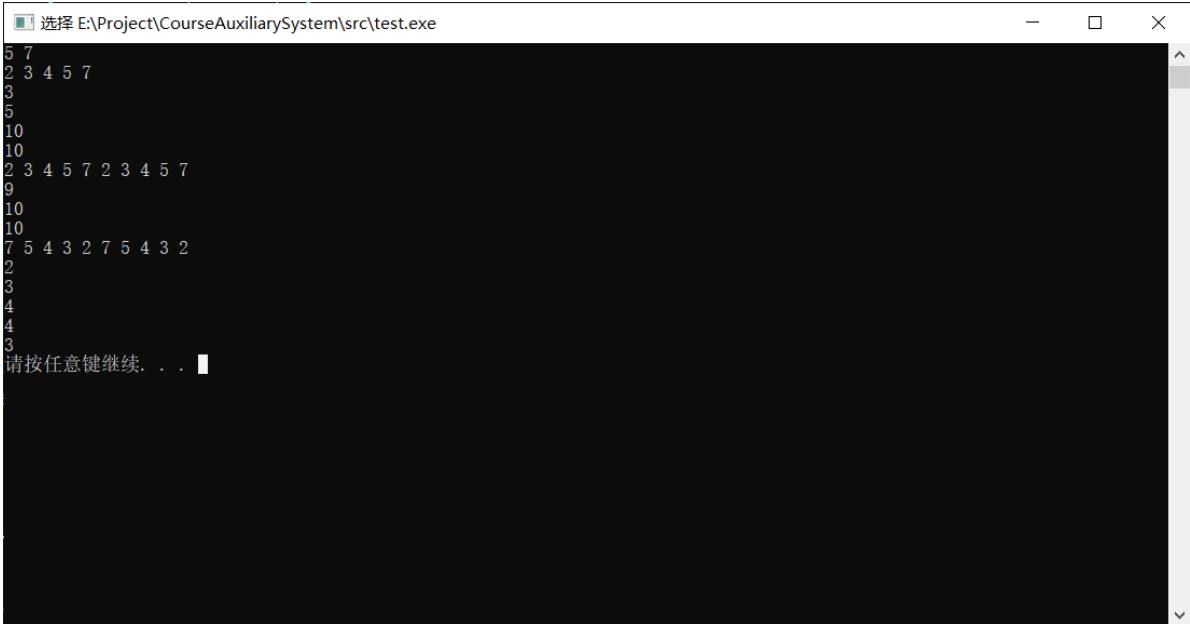
我们以整型数据为例进行了测试，代码如下：

```
Vector<int> A;

int main() {
    int n, Q;
    cin >> n >> Q;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        A.pushBack(x);
    }
    for (int i = 0; i < Q; i++) {
        int x;
        cin >> x;
        if (x == 1) {
            int y;
            cin >> y;
            A.pushBack(y);
        } else if (x == 2) {
            A.popBack();
        } else if (x == 3) {
            cout << A.getSize() << endl;
        } else if (x == 4) {
            cout << A[A.getSize() - 1] << endl;
        } else if (x == 5) {
            A.clear();
        } else if (x == 6) {
            cout << A.isEmpty() << endl;
        } else if (x == 7) {
            int y;
            cin >> y;
            cout << A[y] << endl;
        } else if (x == 8) {
            int y;
            cin >> y;
            A[y] = A[A.getSize() - 1];
            A.popBack();
        } else if (x == 9) {
            A.reverse();
        } else if (x == 10) {
            Vector<int> B = A + A;
            cout << B.getSize() << endl;
            for (int i = 0; i < B.getSize(); i++) {
                cout << B[i] << " ";
            }
        }
    }
}
```

```
        }
    }
    return 0;
}
```

部分测试结果如下图：



```
5 7
2 3 4 5 7
3
5
10
10
2 3 4 5 7 2 3 4 5 7
9
10
10
7 5 4 3 2 7 5 4 3 2
2
3
4
4
3
请按任意键继续. . .
```

3.5 时间类 Time

3.5.1 说明

说明：一个时间类，包含周数、天数和小时数三个值，支持按小时返回值以及时间之间的判等操作。

```
struct Time{
    int week, day, hour;
    Time(int d, int h, int w = 1) : week(w), day(d), hour(h) { }
    Time();
    int calHours();
    bool operator==(const Time& other);
    void operator = (const Time& other);
    void operator = (const int h);
    bool operator < (const Time& other) const;
    bool operator > (const Time& other) const;
};
```

时间类可以与 int 类型通过构造函数、重载运算符 = 和 calHours 函数进行转换，时间类对应的 int 值表示从本学期第 1 天开始计，共经历了多少小时，具体转换规则可以参见如下代码：

```
void operator = (const int h) {
    int hor = h;
    week = hor / (24 * 7) + 1;
    hor = hor % (24 * 7);
    day = hor / 24 + 1;
    hour = hor % 24;
}
int calHours() {
    return ((week - 1) * 7 + day - 1) * 24 + hour;
}
```

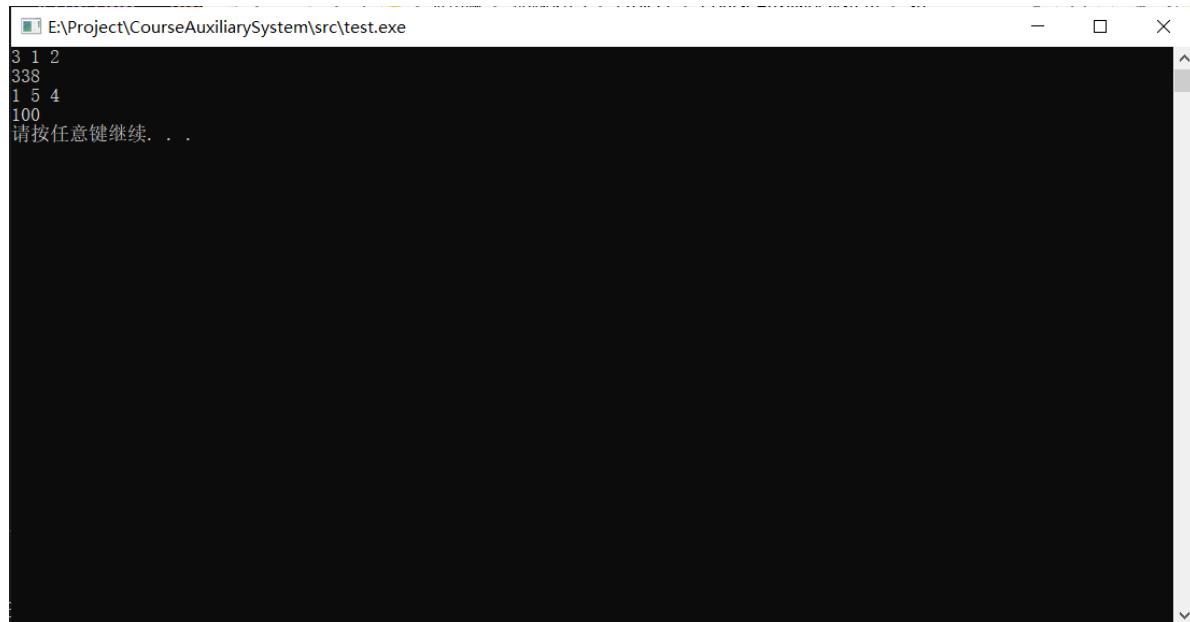
时间类之间重载了大小比较的运算符，比较规则为，以周数 week 为第一关键字、天数 day 为第二关键字、小时数 hour 为第三关键字，比较时间类之间的大小关系。代码比较简单，在这里略过，具体可以参见源代码部分。

3.5.2 测试数据及结果

我们对时间类进行了简单的测试，代码如下：

```
int main() {
    Time x(1, 2, 3);
    cout << x.week << ' ' << x.day << ' ' << x.hour << endl;
    cout << x.calHours() << endl;
    x = 100;
    cout << x.week << ' ' << x.day << ' ' << x.hour << endl;
    cout << x.calHours() << endl;
    return 0;
}
```

运行结果如下：



```
E:\Project\CourseAuxiliarySystem\src\test.exe
3 1 2
338
1 5 4
100
请按任意键继续...
```

4 基础算法

4.1 元素交换、获取最大值

```
namespace Basic {
    template<class T>
    void swapElement(T *x, T *y) {
        T t = *x;
        *x = *y; *y = t;
    }

    template<class T>
    T getMax(T x, T y) {
        if (x > y) return x;
        return y;
    }
} // namespace Basic
```

交换元素函数 swapElement，作用是将两个任意相同类型的元素交换位置，可以自交换。

取最大值函数 getMax，作用是将两个任意相同类型的元素比较，并获得最大值。

4.2 KMP

思想：相比暴力算法kmp是利用之前已经匹配获得的有效信息，比较指针不回溯，而是直接跳过一些字符串（利用next数组），使得原来模式串中的前缀直接移动到后缀位置上，当移动是如果模式串超过被匹配（主串）串范围，则匹配失败。

时间复杂度： $O(m+n)$

算法具体会在课程管理模块结合代码一起分析。

4.3 快速排序

4.3.1 算法描述

1. 从序列中选择一个轴点元素pivot从最后一个元素向前遍历
2. 利用pivot将数组分割成2个子数组：将小于pivot的元素放在pivot的左侧，将大于pivot的元素放在pivot的右侧。将等于pivot的元素任意放于pivot的哪侧都可以。
3. 对子序列进行步骤1和步骤2操作，直到不能再分割(子序列中只剩下一个元素)

```
template<class T>
void Basic::sort(T *a, int len) {
    int i, j;
    T mid = a[(len - 1) / 2];
    i = 0;
    j = len - 1;
    do {
        while (a[i] < mid) i++;
        while (a[j] > mid) j--;
        if (i <= j) {
            T temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
        }
    } while (i < j);
}
```

```

        while (a[j] > mid) j--;
        if (i <= j) {
            swapElement(&a[i], &a[j]);
            i++; j--;
        }
    } while (i <= j);
    if (i < len - 1)
        sort(a + i, len - i);
    if (j > 0)
        sort(a, j + 1);
}

```

这里我们选择的轴点元素是数组的中间元素 mid。每轮排序中，先从左往右找到一个大于 mid 的元素；再从右往左找到小于 mid 的元素，将它们交换位置，直到待交换的两个数的下标交错。在完成此轮交换后，所有比 mid 大的元素都位于 mid 右侧，所有比 mid 小的元素都位于 mid 左侧。整个待排序区间，被分为了 [l, j] 和 [i, r] 两个子区间，子区间之间有序，内部无序。接着递归对两个子区间进行排序即可。

4.3.2 时空复杂度分析

- 空间复杂度：快速排序只使用了一个额外空间进行交换，因此空间复杂度O(1)
- 时间复杂度：快速排序平均用时满足如下递推式：

$$C(n) = (n - 1) + \frac{1}{n} \sum_{j=0}^{n-1} [C(j) + C(n - j - 1)]$$

由主定理可以得到时间复杂度为O(nlogn)

4.3.3 测试用例及结果

对于排序算法，我们以整型数据为例进行了测试，代码如下：

```

int a[103];

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    Basic::sort(a, n);
    for (int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
    system("pause");
}

```

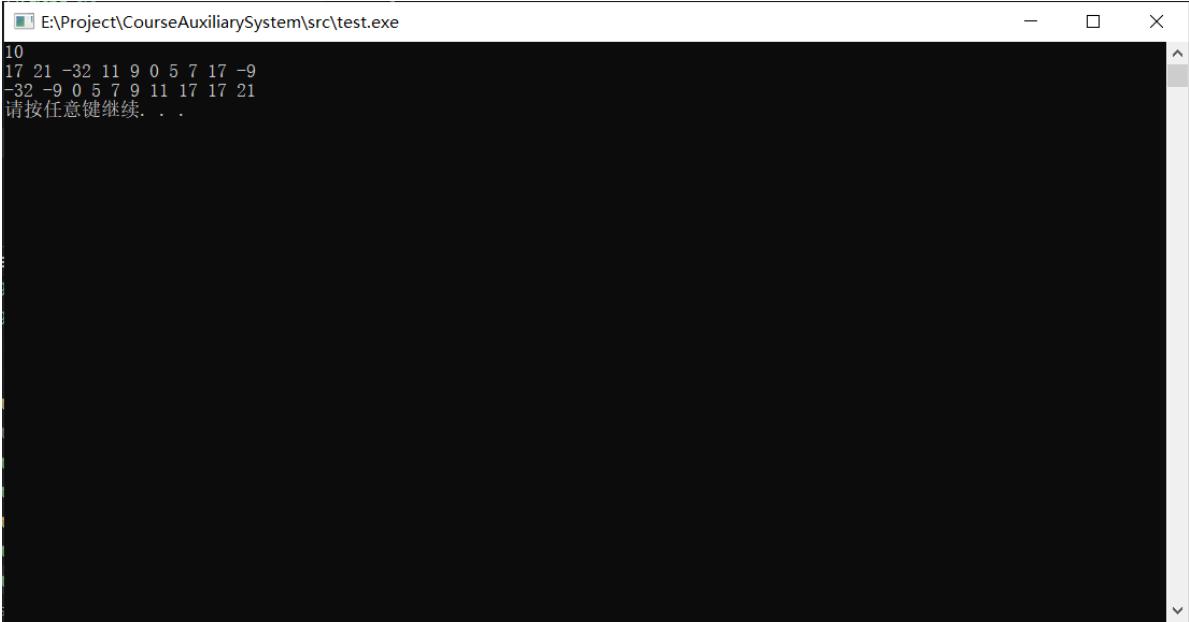
输入数据如下：

```

10
17 21 -32 11 9 0 5 7 17 -9

```

运行结果如下：



A screenshot of a terminal window titled "E:\Project\CourseAuxiliarySystem\src\test.exe". The window contains the following text:

```
10
17 21 -32 11 9 0 5 7 17 -9
-32 -9 0 5 7 9 11 17 17 21
请按任意键继续. . .
```

5 日程管理系统

5.1 系统功能描述

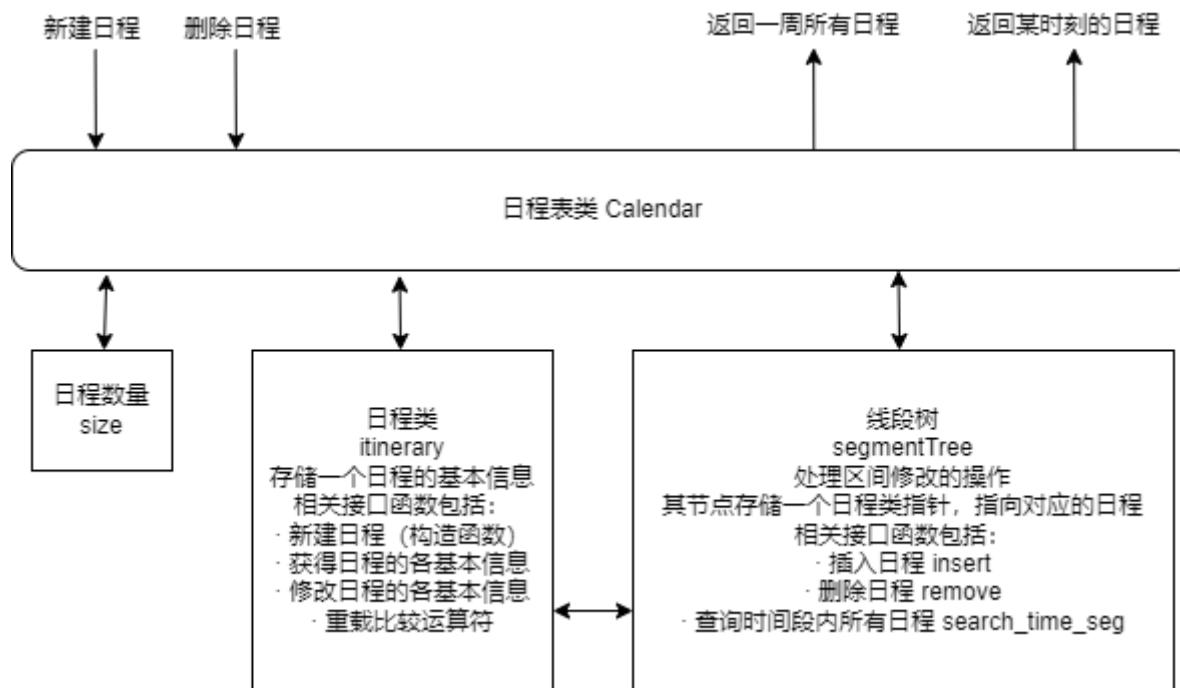
日程管理系统主要维护用户日程安排的日程表。日程表包括每天需要上的课程、自行添加的课外活动、管理员发布的考试三种日程。系统具体功能如下：

- 自动从课程管理系统导入课程的时间安排，并插入日程表中；
- 自动导入管理员发布的考试信息，并插入日程表中；
- 支持用户新建活动日程，并判断是否存在冲突，若不冲突，插入日程表中，否则，发出提示；
- 允许用户对于所有活动进行查找和排序

5.2 系统建模

由上述功能分析可知，日程管理系统需要包含一个存储基本日程（活动、课程或考试）的类，和一个对这些基本日程进行维护的数据结构。在此系统中，我们大量使用了区间操作：判断一个时间区间内是否有日程安排、插入日程并修改一个时间区间、删除日程并清空一个时间区间等，传统算法在处理区间操作时，平均复杂度为 $O(N)$ ，为了提高程序执行效率，我们采用线段树这一数据结构对所有日程进行维护。

5.2.1 系统结构和各函数调用关系



5.2.2 日程类

日程是日程管理系统的基本对象，一个日程对象包括如下信息：

```
class itinerary{
private:
    Pair<Time, Time> t; // 起止时间
    String name; // 日程名称
    int campus; // 所在校区编号
    int location; // 所在建筑地点编号
    int type; // 设置日程类型 (个人、集体、其他 = 0)
}
```

其中 `Pair<Time, Time>` `t` 是一个时间类的二元组，`t.first` 为日程的开始时间，`t.second` 为日程的结束时间，日程的进行时间包含两端点。

日程对象的外部接口有：

```
class itinerary{
public:
    itinerary(); // 构造函数
    itinerary(Pair<Time, Time> tt, String n, int c, int l, int typ = 0);
    // 构造函数
    itinerary(const itinerary &other); // 拷贝构造函数
    ~itinerary(); // 析构函数
    Pair<Time, Time> getTime(); // 获得起止时间
    void setTime(Pair<Time, Time> tt); // 设置起止时间
    String getName(); // 获得名称
    void setName(String n); // 设置名称
    int getCampus(); // 获得校区编号
    int getLocation(); // 获得地点
    int getType(); // 获得类型
    void setLocation(int c, int l); // 设置校区地点
    void setType(int t); // 设置类型
    bool operator < (const itinerary &other) const;
    bool operator > (const itinerary &other) const;
    bool operator == (const itinerary &other) const;
    // 重载比较大小符号，日程大小取决于起止时间的大小
    itinerary operator = (const itinerary &other);
    // 重载赋值
    void print();
    // 日程显示函数，在命令行输出一个日程的基本信息
};
```

主要对日程类每个成员变量的修改和查询，关键函数是对日程类大小关系的重载，即两个日程类的大小关系，取决于它们起止时间的大小关系，开始时间为第一关键字，结束时间为第二关键字。

5.2.3 线段树

算法描述

线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为 $O(\log N)$ 。同时，为了进一步优化效率，只在需要查询的时候进行部分区间的修改而非即可修改整个区间，我们增加了 `lazy` 标记：对整个结点进行的操作，先在结点上做标记，而并非真正执行，直到根据查询操作的需要分成两部分。

类声明：

```
class segmentTree {
// 此线段树对时间区间的操作为左闭右闭，即活动开始时间也不能同结束时间相同
private:
    int lazy[35105];
    struct SegNode {
        bool value;
        bool purity;
        itinerary* point;
```

```

};

SegNode timeSegment[35105]; // 一学期小时数(20*7*24)*4 true代表未被占用
void pushUp(int index);
void pushDown(int index);
// start-end:要查询的区间 curLeft-curRight:
// 当前递归到的区间边界 index:当前时间段对应的数组下标
bool query(int start, int end, int curLeft, int curRight, int index);
// start-end:要查询的区间 curLeft-curRight:当前递归到的区间边界
// state:更新后的状态 index:当前时间段对应的数组下标
void update(int start, int end, bool state, int curLeft, int curRight,
           int index);
void update(int start, int end, bool state, int curLeft, int curRight,
           int index, itinerary* p);

public:
    segmentTree();
    segmentTree(const segmentTree& other);
    bool insert(Pair<Time, Time> t, itinerary* p); // 插入一个新日程，并解决冲突
    void remove(Pair<Time, Time> t); // 删除一个时间段内的所有日程
    void print(int index); // 调试用，从左至右输出整棵线段树的日程，没有去重
    void search_time_seg(int index, int ul, int ur, int l,
                         int r, itinerary* ans, int *asize);
    // 查询接口，可以查询一个时间段内的所有日程，返回ans数组，按时间先后有序
};

```

关键函数分析

(1) 区间查询: bool query(int start, int end, int curLeft, int curRight, int index);

如果当前查询的区间被包含于当前递归到的区间边界，则返回结果，如果尚未被包含于递归到的区间边界，则继续递归查找区间(pushDown)并更新递归区间边界并进行递归查询。

```

bool query(int start, int end, int curLeft, int curRight, int index) {
    if (start <= curLeft && end >= curRight)
        return timeSegment[index];
    else {
        pushDown(index);
        bool res = true;
        int mid = curLeft + ((curRight - curLeft) >> 1);
        if (start <= mid)
            res = res && query(start, end, curLeft, mid, index << 1);
        if (end > mid)
            res = res && query(start, end, mid + 1, curRight, index << 1 |
1);
        return res;
    }
}

```

(2) 区间更新: void update(int start, int end, bool state, int curLeft, int curRight, int index);

对于区间修改，朴素的想法是用递归的方式一层层修改（类似于线段树的建立），但这样的时间复杂度比较高。使用懒标记后，对于那些正好是线段树节点的区间，我们不继续递归下去，而是打上一个标记，将来要用到它的子区间的时候，再向下传递。更新时，我们是从最大的区间开始，递归向下处理。如果更新区间被包括于当前区间，则更新区间状态，如果未包含于当前区间，则传递lazy标记并递归更新左右区间。

```

void segmentTree::update(int start, int end, bool state,
                        int curLeft, int curRight,

```

```

        int index, itinerary* p) {
    // printf("%d %d %d\n", index, curLeft, curRight);
    if (start <= curLeft && end >= curRight) {
        lazy[index] = state ? 1 : -1;
        timeSegment[index].value = state ? 1 : 0;
        timeSegment[index].point = state ? NULL : p;
        timeSegment[index].purity = state ? 0 : 1;
    } else {
        pushDown(index);
        int mid = curLeft + ((curRight - curLeft) >> 1);
        if (start <= mid)
            update(start, end, state, curLeft, mid, index << 1, p);
        if (end > mid)
            update(start, end, state, mid + 1, curRight, index << 1 | 1, p);
        pushUp(index);
    }
}

```

此函数还有一个重载，需要传入一个日程类的指针，在区间更新后，会给线段树每个更新的节点增加一个指向对应日程的指针，算法思路与上面相同。

```

void update(int start, int end, bool state, int curLeft, int curRight,
           int index, itinerary* p) {
    if (start <= curLeft && end >= curRight) {
        lazy[index] = state ? 1 : -1;
        timeSegment[index].value = state ? 1 : 0;
        timeSegment[index].point = state ? NULL : p;
        timeSegment[index].purity = state ? 0 : 1;
    } else {
        pushDown(index);
        int mid = curLeft + ((curRight - curLeft) >> 1);
        if (start <= mid)
            update(start, end, state, curLeft, mid, index << 1, p);
        if (end > mid)
            update(start, end, state, mid + 1, curRight, index << 1 | 1, p);
        pushUp(index);
    }
}

```

(3) 区间插入：bool insert(Pair<Time, Time> t, itinerary* p)

对于一个新日程，插入时应取出它的起止时间，接着询问该时间区间内是否已经存在其他日程：若存在，则说明发生了日程冲突，立即退出，并返回 False；否则修改该区间状态为占用，并为线段树对应节点添加一个指针。

线段树的 insert 函数是外部直接调用的接口函数之一，传入一个占用时间段 t，和一个对应日程的指针 p。线段树首先对时间段 t 进行一次 query，询问这个时间段内是否被占用，如果被占用，返回 false。

如果没有被占用，线段树则调用一次 update 函数，对这个时间段进行更新，更新状态为占用，添加日程指针为 p。返回 true。

```
bool segmentTree::insert(Pair<Time, Time> t, itinerary* p) {
    int start = t.first.calHours();
    int end = t.second.calHours();
    // printf("%d %d\n", start, end);
    if (!query(start, end, 1, 8760, 1)) {
        // printf("*1\n");
        return false;
    } else {
        // printf("Ready to insert!\n");
        update(start, end, false, 1, 8760, 1, p);
        return true;
    }
}
```

(4) 区间删除: void remove(Pair<Time, Time> t)

线段树的 `remove` 函数是外部直接调用的接口函数之一，它的作用是将一个时间段设置为非占用状态。直接调用 `update` 函数，设置参数为 0 即可。

```
void segmentTree::remove(Pair<Time, Time> t) {
    int start = t.first.calHours();
    int end = t.second.calHours();
    update(start, end, true, 1, 8760, 1);
}
```

(5) 时间段查询: search_time_seg(int index, int ul, int ur, int l, int r, itinerary* ans, int* asize)

线段树的外部接口函数之一，作用是查询一段时间内包含哪些日程，按照时间先后顺序返回到数组 ans 中。其中参数 index, ul, ur 是递归使用的参数，分别表示当前线段树上的节点编号，当前节点维护区间的左端点，当前节点维护区间的右端点；参数 l, r 是外部传入的查询时间段；参数 ans 和 asize 是传指针的外部数组，用来返回查询答案，ans 是数组起始位置的指针，asize 是数组元素个数。

调用此函数时，从根节点 1 开始搜索，根节点维护的时间段范围为 1 - 8760. 此函数首先判断当前节点维护的区间，是否在查询范围内。若超出查询范围，直接返回。

接着它会判断当前节点维护的时间区间是否属于一个完整的日程。根据之前 update 函数的定义，如果一个节点维护的区间属于一个完整的日程，那么这个节点会存储一个 point 指针指向该日程；否则，这个节点的 point 指针值为 NULL。那么此函数会判断当前节点的 point 是否为 NULL，若非空，说明这个节点属于一个完整的日程，那么判断这个节点对应的日程是否已经被添加进 ans 数组里，若没有，则添加入 ans 数组中。

如果这个节点 point 指针值为 NULL，说明这个节点不属于一个完整的日程，接着递归询问此节点的左右区间。

需要注意的是，这里的 point 指针起到了类似于传统线段树懒惰标记的作用，当我们访问到 point 非空的节点，说明此节点的所有子树节点都属于同一个日程了，直接保存答案返回即可，不用再去访问它的全部子节点。正是这种优化，保证了它查询具有均摊 \log 的复杂度。

不过，就算我们已经用懒惰标记避免了对子树的查询，依然可能出现日程重复的情况。不过，因为我们代码实现中，是严格先序遍历整棵线段树的，而线段树的子树天然按照区间从左至右有序，所以我们取得的日程按照时间是严格不降的。因此，每次我们判断日程是否重复时，只需要和 `ans` 数组保存的上一个日程比较是否相同即可，不用把 `ans` 数组整个遍历一遍判重，可以节约很多时间。是一个利用线段树性质的小技巧。这个性质也保证了我们返回 `ans` 数组是按照时间先后严格有序的。

```

if (ur < l || ul > r) return;
// printf("%d %d %d %d %d\n", index, ul, ur, l, r);
if (timeSegment[index].point) {
    // printf("!!! %d %d %d %d %d\n", index, ul, ur, l, r);
    if ((*asize) == 0 ||
        !(ans[(*asize) - 1] == *timeSegment[index].point)) {
        // printf("Begin to add\n");
        ans[(*asize)] = *timeSegment[index].point;
        // (*timeSegment[index].point).print();
        // ans[(*asize)].print();
        (*asize) = (*asize) + 1;
        // printf("Ended!\n");
    }
    return;
} else {
    if (ul == ur) return;
    int mid = ul + ((ur - ul) >> 1);
    // printf("? %d %d %d %d %d", ul, ur, ul, mid, mid + 1, ur);
    if ((index << 1) > 35100) return;
    search_time_seg(index << 1, ul, mid,
                     l, r, ans, asize);
    if ((index << 1 | 1) > 35100) return;
    search_time_seg(index << 1 | 1, mid + 1, ur,
                     l, r, ans, asize);
}

```

(6) 测试输出: printf(int index)

这是测试线段树正确性的调试用函数，可以输入一个节点的序号，接着按照时间先后顺序，依次输出以该节点为根节点的子树内所有日程。这里没有添加去重功能，仅是为了方便观察线段树的结构和检验正确性。

```

void segmentTree::print(int index) {
    // for(int i = 1; i < 20001; i++)
    //     printf("%d %d\n", i, timeSegment[i].purity);
    // printf("index: %d purity: %d\n", index, timeSegment[index].purity);
    // printf("%d ", index);
    if (timeSegment[index].purity) {
        itinerary* p = timeSegment[index].point;
        p->print();
        // printf("Ended!\n");
    } else {
        if ((index << 1) > 35100) return;
        print(index << 1);
        if ((index << 1 | 1) > 35100) return;
        print(index << 1 | 1);
    }
}

```

时间复杂度分析

线段树考虑线段树进行区间操作时的过程，如果当前节点被操作区间完全包含，则不会继续向下递归。考虑对区间 L, R 进行操作，若当前节点 $mid < L$ ，则会向右子树递归，若 $mid \geq R$ ，则会向左子树递归。若 $L \leq mid < R$ ，则区间会分为两部分分别向左右子树递归，即区间 $[L, mid]$ 和 $[mid + 1, R]$ 。两部分情况相似，我们先考虑右半边。若当前节点 $mid \geq R$ ，则会向左子树递归。否则，只会向右子树递归，因为此节点的左孩子一定被 $[L, R]$ 完全覆盖。左半边同理。因此，对区间操作

时，最多只会递归 $2Deep$ 次，其中 $Deep$ 为线段树的深度。线段树的深度是 $O(\log n)$ 的，因此单次区间操作的复杂度也为 $O(\log n)$.

测试用例及结果

我们单独给线段树模块进行了测试，测试代码如下：

```
segmentTree A;
itinerary x[103];

int main() {
    int Q;
    cin >> Q;
    for (int i = 0, l, r; i < Q; i++) {
        int op;
        Pair<Time, Time> t;
        cin >> op;
        if (op == 1) {
            char str[103];
            cin >> l >> r >> str;
            t.first = l;
            t.second = r;
            String y;
            int m = strlen(str);
            for (int i = 0; i < m; i++)
                y.pushBack(str[i]);
            x[i].setName(y);
            x[i].setTime(t);
            if (A.insert(t, x + i))
                puts("Yes!");
            else
                puts("No!");
        } else if (op == 2) {
            cin >> l >> r;
            t.first = l;
            t.second = r;
            A.remove(t);
        } else if (op == 3) {
            A.print(1);
        }
    }
    system("pause");
    return 0;
}
```

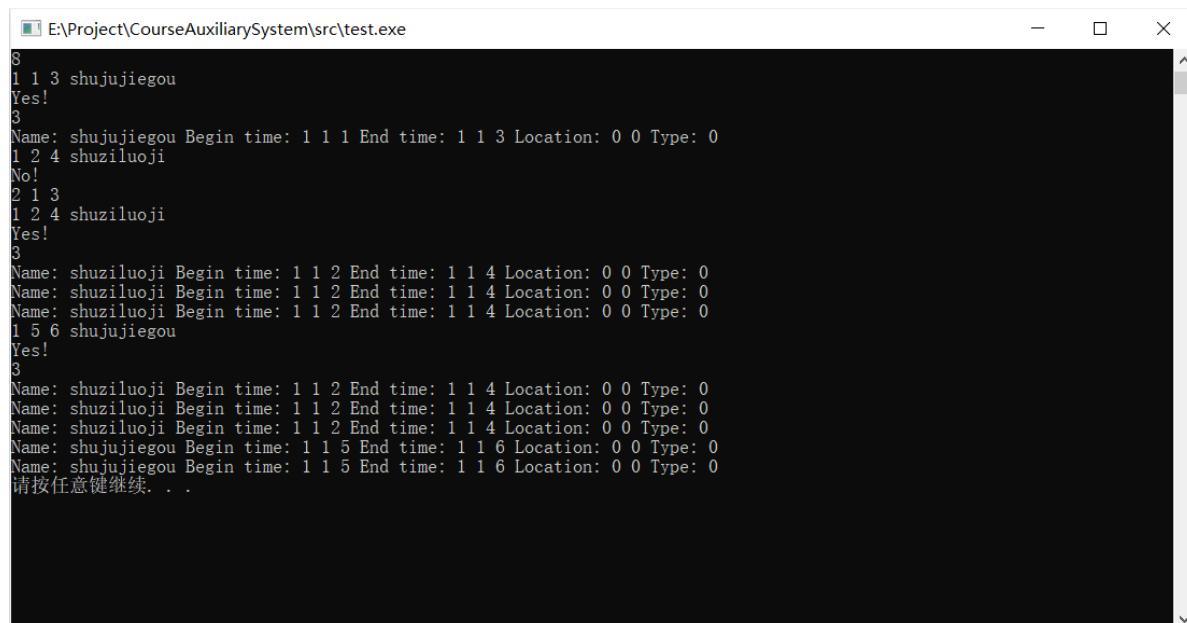
输入：

```
8
1 1 3 shujujiegu
3
1 2 4 shuziluoji
2 1 3
1 2 4 shuziluoji
3
1 5 6 shujujiegu
3
```

输出：

Yes!
Name: shujujiegou Begin time: 1 1 1 End time: 1 1 3 Location: 0 0 Type: 0
No!
Yes!
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Yes!
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shuziluoji Begin time: 1 1 2 End time: 1 1 4 Location: 0 0 Type: 0
Name: shujujiegou Begin time: 1 1 5 End time: 1 1 6 Location: 0 0 Type: 0
Name: shujujiegou Begin time: 1 1 5 End time: 1 1 6 Location: 0 0 Type: 0

运行结果：



5.2.4 日程表类 Calendar

日程表类是日程管理系统最外层的封装，它本身即是一个实现了完整功能的模块，留有丰富的功能接口，直接与前端对接。

日程表类具体定义如下：

```

// 新建日程的话，使用这个接口可以新建 + 插入线段树
bool insert(itinerary* x); // 如果 itinerary 已经有的话，可以用这个
void deleteItinerary(itinerary* x);
void print();
Vector<itinerary> show_week(int week);
// 输入周数，把本周的所有日程返回，返回值是日程的vector
Vector<itinerary> show_hour(int time);
// time = (week - 1) * 24 * 7 + (day - 1) * 24 + hour
};

```

它包括两个成员变量，一个是维护所有日程的线段树，一个是记录已经有的日程数量的size。

对外部的接口函数包括：

- isEmpty() 判断日程表是否为空
- newItinerary 传入日程的基本信息，新建一个日程，并插入线段树中
- insert 传入一个日程指针，将该日程插入线段树中
- print 测试用函数，作用是把线段树输出一遍
- show_week 输入周数，返回本周的所有日程
- show_hour 输入一个时间，返回该时间对应的日程

关键函数如下：

(1) 新建日程：itinerary* newItinerary(Pair<Time, Time> tt, String n, int c, int l, int typ)

新建日程函数会根据输入的参数，动态申请一个日程类 itinerary，然后调用 insert 函数尝试将该日程插入线段树中。如果 insert 函数返回 0，表示插入失败，那么释放掉申请的新日程类，返回空指针；否则返回新日程类的指针。

```

itinerary* Calendar :: newItinerary(Pair<Time, Time> tt,
                                      String n, int c, int l, int typ) {
    itinerary* x = new itinerary;
    x->setLocation(c, l);
    x->setTime(tt);
    x->setName(n);
    x->setType(typ);
    if (x == NULL) return x;
    if (!insert(x)) return x;
    delete x;
    x = NULL;
    return x;
}

```

(2) 插入日程：bool insert(itinerary* x)

传入一个日程类的指针 x，表示待插入日程的指针。调用线段树的 insert 函数，传入日程类 *x 的时间段和指针，如果插入失败返回 0，如果插入成功返回1.

```

bool Calendar :: insert(itinerary* x) {
    Pair<Time, Time> t = x->getTime();
    if (!seg.insert(t, x)) return 0;
    size++;
    return 1;
}

```

(3) 删除日程：void deleteItinerary(itinerary* x)

传入一个日程类的指针 x，表示待删除日程的指针。调用线段树的 remove 函数，将该日程占用时间段标记为空，日程数量 size - 1，释放该日程的指针。

```
void Calendar :: deleteItinerary(itinerary* x) {
    Seg.remove(x->getTime());
    size--;
    delete x;
    x = NULL;
}
```

5.3 模块测试用例及测试结果

日程表可视化

主要通过show_week接口进行查询，并通过相关可视化函数实现，结果如下图所示：

日程表		周数: 24_6月_14日 星期2					
		活动查询					
		● 个人 ○ 集体					
Mon.	Tue.	Wed.	Thur.	Fri.	Sat.	Sun.	
8:00 计算机网络 沙河校区 教学实验综合楼 S516 课程或考试	形式语言与自动机 沙河校区 教学实验综合楼 S510 课程或考试	计算机网络课程设计 沙河校区 教学实验综合楼 N210 课程或考试		高等数学 西土城校区 主楼 N110 课程或考试			
9:00							
10:00		数字逻辑课程设计 沙河校区 教学实验综合楼 N210 课程或考试	数据结构课程设计 沙河校区 教学实验综合楼 N214 课程或考试				
11:00 计算机组成原理 沙河校区 教学实验综合楼 S510 课程或考试				线性代数 西土城校区 主楼 N508 课程或考试			
12:00							
13:00							

插入日程并检测冲突

输入：

UI

是否设置活动闹钟

Activity Name 模块测试

Loc 图书馆

Location: 一层

Date: 2022/6/16

Start Time 19:00

End Time 20:00

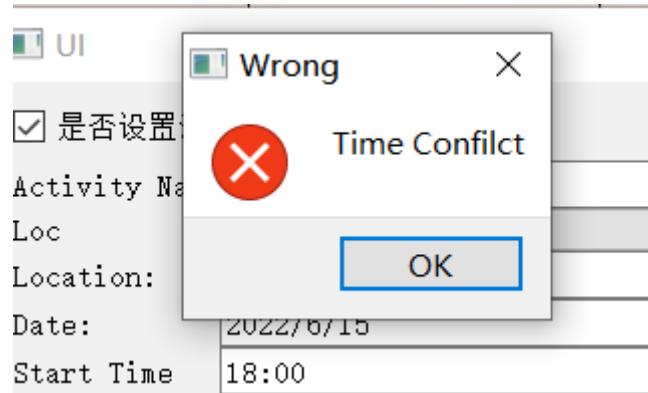
private activity

group activity

OK

Cancel

输出 (时间冲突) :



输出 (成功创建并可视化) :



6 校园导航系统

6.1 系统功能描述

校园导航系统主要处理用户查询的导航需求，需要考虑跨校区通勤、是否骑乘自行车、最短距离或最少时间等情况，具体功能如下：

- 导入校园地图
- 求出两点间最短距离
- 求出两点间骑车或走路的最短时间
- 给出起点、终点以及一系列中间点，求最短路径
- 支持跨校区导航

6.2 系统建模

由上述功能分析可知，校园导航系统需要实现最短路径的求解。对于校区内部的最短路径，我们可以采用传统的最短路径算法，如 Dijkstra、Floyd等算法。对于跨校区的最短路径，我们遍历所有跨校区通勤方式，分别求出从起点到上车点、下车点到终点的最短路径，再二分求出乘坐班车的发车时间。

为了保证逻辑结构的清晰，我们将整个校园导航系统分成三个类：地图类、校区类、导航类

6.2.1 堆

算法描述

堆是一个关键字较大（或较小）元素先出的优先队列，它是一棵二叉树，每一个节点都大于（或小于）它的两个孩子节点。堆支持插入、删除、取堆顶元素等操作。其定义如下：

```
template <class T, typename F = Less<T> >
class Heap {
private:
    vector<T> data;
    int size;
    void swap(int x, int y);

public:
    Heap();
    void push(T x);
    T top() const;
    bool isEmpty() const;
    int getSize() const;
    void pop();
};
```

堆是一个模板类，需要传入数据类型 T，和比较方法 F。默认比较方法 F 是小于，其他比较方法定义及实现如下：

```
template<class T>
struct Less {
    bool operator() (const T &lhs, const T &rhs) const;
};

template<class T>
struct Greater {
```

```

    bool operator() (const T &lhs, const T &rhs) const;
};

template<class T>
struct LessEqual {
    bool operator() (const T &lhs, const T &rhs) const;
};

template<class T>
struct GreaterEqual {
    bool operator() (const T &lhs, const T &rhs) const;
};

```

堆有两个成员函数。其中，变长数组 data 即为堆中元素所在的数组，是堆实际占用的空间。整形 size 为堆的大小。内部函数 swap 提供堆中两个元素的交换方法。

对外的接口函数包括：将元素插入堆 push，取堆顶元素 top，判空函数 isEmpty，取堆大小 getSize，弹出堆顶元素 pop。下面具体分析比较重要的几个函数：

(1) 插入函数 push

插入函数可以将一个元素 x，插入堆中，并调整它在堆中的位置，保证插入完成后，堆依然是一个满足性质的二叉树。其插入过程分为两步：

- 插入元素初始放在堆尾，堆的长度加一；
- 对插入元素进行“上浮”。插入元素与其父亲结点比较大小，若比其父亲结点大（或小），则与其父亲结点交换。再考查此结点与其新的父亲结点的大小关系，直至到达堆顶。

代码实现如下：

```

template <class T, typename F>
void Heap<T, F>::push(T x) {
    size++;
    data.pushBack(x);
    int i = size;
    while (i > 1 && F()(data[i >> 1], data[i])) {
        swap(i, i >> 1);
        i = i >> 1;
    }
}

```

(2) 取堆顶函数 top

堆顶就是二叉树的根节点，因此只需要取出当前根节点对应的数据即可，代码实现如下：

```

template <class T, typename F>
T Heap<T, F>::top() const {
    return data[1];
}

```

(3) 弹出堆顶元素函数 pop

弹出当前堆顶的元素，并调整堆中其余元素的位置，保证弹出之后，堆依然是一个符合条件的二叉树。弹出操作分为两步：

- 用堆尾元素替换堆顶元素，size - 1
- 对此元素进行“下沉”操作。每次比较当前元素和左右孩子的大小关系，若比某左右孩子的最大值小（或大），则与此最大值对应的左右孩子进行交换。直至比左右孩子都要大（或小）。

代码实现如下：

```
template <class T, typename F>
void Heap<T, F>::pop() {
    if (size == 0) return;
    data[1] = data[size];
    data.popBack();
    size--;
    int i = 1, j;
    if ((i << 1) > size) return;
    if (((i << 1) | 1) > size || F()(data[(i << 1) | 1], data[i << 1]))
        j = i << 1;
    else
        j = (i << 1) | 1;
    while (j <= size && F()(data[i], data[j])) {
        swap(i, j);
        i = j;
        if ((i << 1) > size) return;
        if (((i << 1) | 1) > size || F()(data[(i << 1) | 1], data[i << 1]))
            j = i << 1;
        else
            j = (i << 1) | 1;
    }
}
```

其余函数较为简单，在此处略去，具体参见源程序部分。

时空复杂度分析

- 空间复杂度：堆排序只使用了一个额外空间进行交换，因此空间复杂度 $O(1)$
- 时间复杂度：堆为一棵完全二叉树，因此其树高不超过 $O(\log n)$ ，每次插入删除会从叶子走到根或从根走到叶子，因此单次插入删除的时间复杂度为 $O(\log n)$ ，总共会进行 n 次插入与删除，因此总时间复杂度为 $O(n \log n)$

测试用例及结果

我们以整型数据为例对堆进行了测试，代码如下：

```
Heap<int> heap;

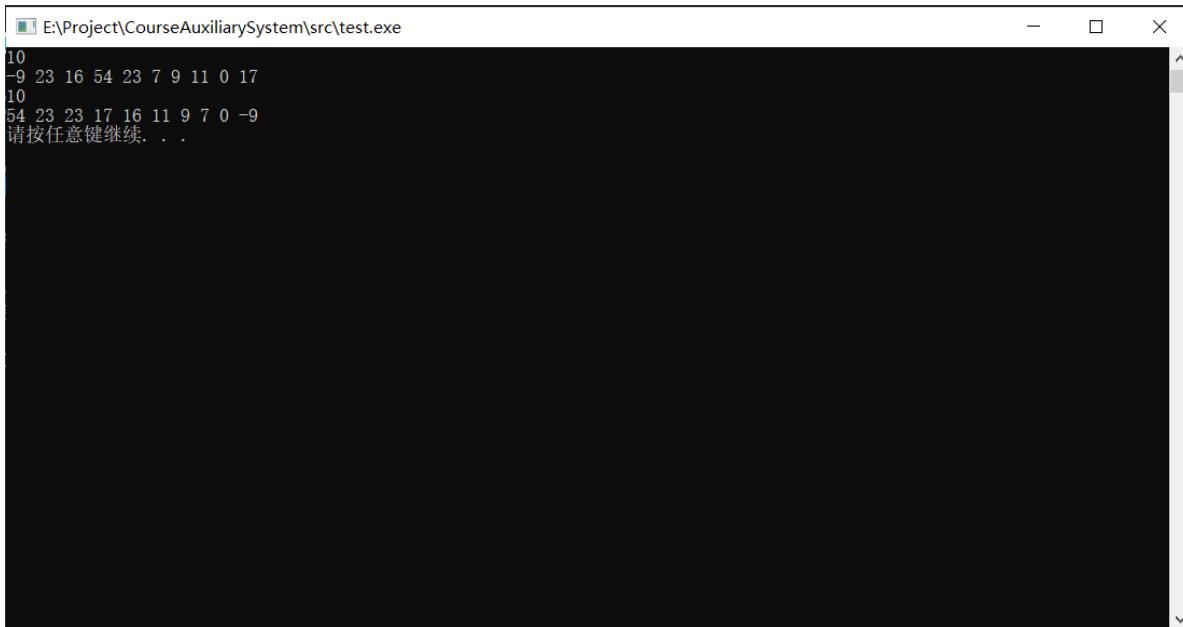
int main() {
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        heap.push(x);
    }
    cout << heap.getSize() << endl;
    for (int i = 1; i <= n; i++) {
        cout << heap.top() << " ";
        heap.pop();
    }
    cout << endl;
    return 0;
}
```

输入：

```
10
-9 23 16 54 23 7 9 11 0 17
```

运行结果：



```
E:\Project\CourseAuxiliarySystem\src\test.exe
10
-9 23 16 54 23 7 9 11 0 17
10
54 23 23 17 16 11 9 7 0 -9
请按任意键继续. . .
```

6.2.2 最短路径算法

边类与地图类

以校园内每个建筑物或关键点为顶点集，校园内所有道路为边集，我们就构建了一个图 $G(V, E)$ 。对于这个图上的所有边，我们定义了一个边类 EdgeNode：

```
struct EdgeNode {
    double dis;
    double crowdDegree;
    bool isBike;
    EdgeNode();
    EdgeNode(double d, double c, bool f);
};
```

其中，`dis` 表示这条道路的长度，`crowdDegree` 表示道路的拥挤程度，`isBike` 表示道路是否允许自行车通行。

地图类用于处理一个抽象的图上的最短路径问题，声明如下：

```
typedef vector<pair<EdgeNode, int>> Route;

class Graph {
private:
    Array<int> head;
    Vector<int> end, next;
    Vector<EdgeNode> weight;
    int vertexNum, edgeNum;
```

```

public:
    Graph();
    explicit Graph(int n);
    ~Graph();
    void addDirectedEdge(int u, int v, EdgeNode w);
    void addUndirectedEdge(int u, int v, EdgeNode w);
    template<typename F>
    Route* shortestPath
        (int startVertex, int endVertex, double *ans);
    template<typename F>
    Route shortestPath
        (int startVertex, int endVertex, Vector<int> midVertex, double *ans);
    template<typename F>
    Array<double> singleSourceShortestPath(int startVertex);
};


```

边权计算

在计算最短路径的时候，我们会面临多种情况，即是否骑乘自行车、最短距离或最短时间等。朴素的想法是对每种情况都建立一个图，这样会导致大量的空间浪费，代码实现上也十分冗长。因此，我们充分利用了 C++ 语言的特性，在求解最短路时传入边权计算函数 F 。不难发现，不管是哪种情况，都只涉及边权的计算，即便骑乘自行车的情况会导致一些边被禁用，我们也只需将这些边的边权设为无穷大即可。边权计算类的定义如下：

```

struct WalkCalc {
    double operator () (const EdgeNode &x) const {
        return walkSpeed * x.crowdDegree < 1e-9 ?
            1e18 : x.dis / (walkSpeed * x.crowdDegree);
    }
};

// 步行最短时间

struct BikeCalc {
    double operator () (const EdgeNode &x) const {
        return x.isBike ? (bikeSpeed * x.crowdDegree < 1e-9 ?
            1e18 : x.dis / (bikeSpeed * x.crowdDegree)) : 1e18;
    }
};

// 骑车最短时间

struct DisCalc {
    double operator () (const EdgeNode &x) const {
        return x.dis;
    }
};

// 最短距离

```

两点间最短路径

在求解两点间最短路径时，我们采用了堆优化 Dijkstra 算法。传统的 Dijkstra 算法可以求解单源最短路径长度，但导航时我们需要将具体路径求出来。我们只需在更新最短路径时，记录当前点最短路径是从哪个点转移过来即可，即从起点到当前点最短路径上的所有前驱，最后从终点 dfs 求出所有的最短路径。在这里，考虑到实际应用场景与内存占用，如果有多条最短路径，我们至多会保留 10 条。实现如下：

```

Route* shortestPath(int startVertex, int endVertex, double *ans) {
    for (int i = 0; i < 10; i++)
        routes[i].clear();
    routeNum = 0;
    if (startVertex == endVertex) {
        *ans = 0;
        return routes;
    }
    Array<double> dis(vertexNum + 1, 1e18);
    Heap<Pair<double, int>, Greater<Pair<double, int>> Q;
    Q.push(Pair<double, int> (0, startVertex));
    dis[startVertex] = 0;
    int tot = 0;
    while (!Q.isEmpty()) {
        auto tmp = Q.top();
        Q.pop();
        if (dis[tmp.second] != tmp.first)
            continue;
        for (int index = head[tmp.second]; index != 0; index = next[index])
            if (dis[end[index]] > F()(weight[index]) + (tmp.first)) {
                int v = end[index];
                dis[v] = F()(weight[index]) + (tmp.first);
                pre[v].clear();
                preEdge[v].clear();
                pre[v].pushBack(tmp.second);
                preEdge[v].pushBack(weight[index]);
                Q.push(Pair<double, int> (dis[end[index]], end[index]));
            } else if (dis[end[index]] == F()(weight[index]) + (tmp.first)) {
                int v = end[index];
                pre[v].pushBack(tmp.second);
                preEdge[v].pushBack(weight[index]);
            }
        }
        *ans = dis[endVertex];
        dfs(startVertex, endVertex, Route());
        return routes;
    }

    void dfs(int startVertex, int curVertex, Route x) {
        if (routeNum >= 10)
            return;
        if (curVertex == startVertex) {
            x.reverse();
            routes[routeNum++] = x;
            return;
        }
        for (int i = 0; i < pre[curVertex].getSize(); ++i) {
            Route y = x;
            y.pushBack(Pair<EdgeNode, int> (preEdge[curVertex][i], curVertex));
            dfs(startVertex, pre[curVertex][i], y);
        }
    }
}

```

在存储路径时，我们使用了 `Pair<EdgeNode, int>` 表示从上一个点到当前点经过的边以及当前点的编号。之所以将具体的边的信息保存，是考虑到两点间可能存在多条路径的情况，方便加以区分。

经过多点最短路径

在处理途经多点的最短路径时，我们首先计算出任意两点的最短路径。在处理好任意两点间最短路径后，我们采用状态压缩动态规划的算法。记 $F[S][u]$ 表示当前经过点的集合为 S ，且最后到达的点为 u 的最短路径长度，则有 $F[S \cup v][v] = \min F[S][u] + dis[u][v]$

由此可以求解出途经多点的最短路径。代码实现如下：

```
template<typename F>
Vector<Pair<EdgeNode, int>> Graph :: shortestPath(
    int startVertex,
    int endVertex,
    Vector<int> midVertex,
    double *ans
) {
    int num = midVertex.getSize();
    int maxVertex = num + 1;
    const int maxState = (1 << num) | 1;
    DyadicArray<double> f(maxState, maxVertex, 1e18);
    DyadicArray<int> pre(maxState, maxVertex, 0);
    Array<double> startDis = singleSourceShortestPath<F>(startVertex);
    Array<Array<double>> dis(num);
    for (int index = 0; index < num; index++) {
        dis[index] = singleSourceShortestPath<F>(midVertex[index]);
        f[1 << index][index] = startDis[midVertex[index]];
    }
    // 动态规划初始化
    for (int state = 0; state < (1 << num); state++) {
        for (int index = 0; index < num; index++) {
            if (~(state >> index) & 1) continue;
            double w = f[state][index];
            if (w > 1e9) continue;
            for (int vertex = 0; vertex < num; vertex++) {
                if ((state >> vertex) & 1) continue;
                int newState = state | (1 << vertex);
                if (f[newState][vertex] > w + dis[index][midVertex[vertex]]) {
                    f[newState][vertex] = w + dis[index][midVertex[vertex]];
                    pre[newState][vertex] = index;
                }
            }
        }
    }
    double ansDis = 1e18;
    Array<int> nextVertex(vertexNum + 1);
    Vector<Pair<EdgeNode, int>> path;
    int tmp;
    for (int vertex = 0; vertex < num; vertex++) {
        if (ansDis > f[(1 << num) - 1][vertex] + dis[vertex][endVertex]) {
            ansDis = f[(1 << num) - 1][vertex] + dis[vertex][endVertex];
            tmp = vertex;
        }
    }
    *ans = ansDis;
    nextVertex[midVertex[tmp]] = endVertex;
    int state = (1 << num) - 1;
    while (state != (1 << tmp)) {
        int p = pre[state][tmp];
        nextVertex[midVertex[p]] = midVertex[tmp];
        state ^= (1 << tmp);
        tmp = p;
    }
}
```

```

    }
    path = shortestPath(startVertex, midVertex[tmp]);
    for (tmp = midVertex[tmp]; tmp != endVertex; tmp = nextVertex[tmp])
        path = path + shortestPath(tmp, nextVertex[tmp]);
    return path;
}

```

6.2.3 校区类

在地图类的基础上，我们对每个校区建立了一个校区类 Campus，主要用于处理外部的传参，定义如下：

```

struct Campus {
    Graph G;
    Campus();
    explicit Campus(int n);
    ~Campus();
    void addDirectedEdge
        (int u, int v, double dis, double crowdDegree, bool bikeAccess);
    void addUndirectedEdge
        (int u, int v, double dis, double crowdDegree, bool bikeAccess);
    Route* shortestPath
        (int startVertex, int endVertex, double *ans, int type);
    Array<double> singleSourceShortestPath(int startvertex, int type);
    Route shortestPath
        (int startVertex, int endVertex, Vector<int> midvertex, double *ans, int
type);
};

```

该类主要是为了保证逻辑上的清晰和方便外部调用，处理了传入边权计算类的部分。

6.3.4 导航类

导航类是整个模块最外层、直接调用的部分，主要处理了跨校区的导航。定义如下：

```

typedef Vector<Pair<EdgeNode, int>> Route;

struct CrossEdge {
    int x, y, u, v;
    double dis, time;
};

struct CrossTimeNode {
    CrossEdge e;
    int m;
    Vector<double> timeTable;
};

class Guider {
private:

```

```

int campusNum, crossEdgeNum;
vector<CrossTimeNode> crossEdge;
Campus campusMap[2];

public:
    Route tmpx[11], tmpy[11];
    int xNum, yNum;
    double busTime;
    CrossEdge busEdge;
    Guider();
    Guider(int n, int m1, int m2);
    void addEdge(int x, int u, int v, double d, double c, bool f);
    void addEdge(CrossTimeNode w);
    double shortestPath(
        int startCampus,
        int startVertex,
        int endCampus,
        int endVertex,
        double curTime,
        int type);
    double shortestPath(
        int startCampus,
        int startVertex,
        int endCampus,
        int endVertex,
        double curTime,
        Vector<int> midVertex1,
        Vector<int> midVertex2,
        int type);
};


```

在这里，我们用 CrossTimeNode 类存储跨校区通行方式，包括了基本属性以及发车时间列表。在处理跨校区导航时，我们遍历所有跨校区通行方式，求出从起点到上车点花费的时间，再根据当前时间二分求出乘坐班车的发车时间。

6.3 模块测试用例及测试结果

在实际应用校园导航时，我们对真实校园地图进行了建模。分为西土城校区与沙河校区，校区内的每条边用一个五元组 (u, v, d, f, r) 表示，指存在一条从 u 到 v 长度为 d 的边， $f=1$ 表示可以自行车通行。对于跨校区的通行方式，我们用 $(x, u, y, v, t, TimeTable)$ 表示从 x 校区点 u 到 y 校区点 v 存在耗时为 t ，时刻表为 $TimeTable$ 的通行方式。我们对校区内、校区间、不同通行方式等导航策略均进行了测试。

输入：

```

1 1
0 1 1 27 9
2 2 3
4 1 2 3 4

```

该测试数据表示导航策略为经过多点、步行最短时间，从校区 0 的 1 号点途经 2、3 号点到达校区 1，途经校区 1 的 1、2、3、4 号点到达 27 号点，出发时间为 9 点。

输出:

```
119.31
=====
In Campus 0:
1 -> 0 -> 2 -> 3 -> 4 -> 35 -> 14
From Campus 0 to 1:
10:0 14 -> 27
In Campus 1:
27 -> 25 -> 34 -> 36 -> 38 -> 39 -> 5 -> 1 -> 4 -> 2 -> 3 -> 45 -> 46 -> 29 ->
27
=====
```

程序输出了一条最短路径，在校区0内路径为1->0->2->3->4->35->14，随后在14号点乘10:00的车前往校区1 27号点，校区1内路径亦显示在输出中了。可以发现均包含了中间点。

输入:

```
0 0
0 7 0 19 9
```

该数据表示导航策略为经过多点、最短距离，从校区0的7号点到达校区0的19号点。

输出:

```
82
=====
Route #1:
In Campus 0:
7 -> 36 -> 19
=====
```

输出表示最短距离为82m，仅有一条路径。

输入:

```
0 0
0 28 0 33 0
```

输出:

```
=====
Route #1:
In Campus 0:
28 -> 3 -> 4 -> 33
=====
=====
Route #2:
In Campus 0:
28 -> 3 -> 21 -> 33
=====
```

此时输出了两条路径

7 课程管理系统

7.1 系统功能描述

课程管理系统主要维护用户的课程信息和课程资料的上传下载。课程管理包括课程的所有基本信息，课程的考试安排，和所有对课程的添加、删除、修改、查询等操作；文件管理包括所有课程的上传资料和作业，支持添加、删除、查找、排序、查重等操作。系统具体功能如下：

- 自动导入管理员发布的课程信息
- 自动导入管理员发布的考试信息
- 支持修改课程的基本信息
- 支持用户在课程表上查看课程的相关信息
- 支持用户上传、压缩、下载课程资料或作业
- 支持用户上传资料作业的查重、去重

7.2 系统建模

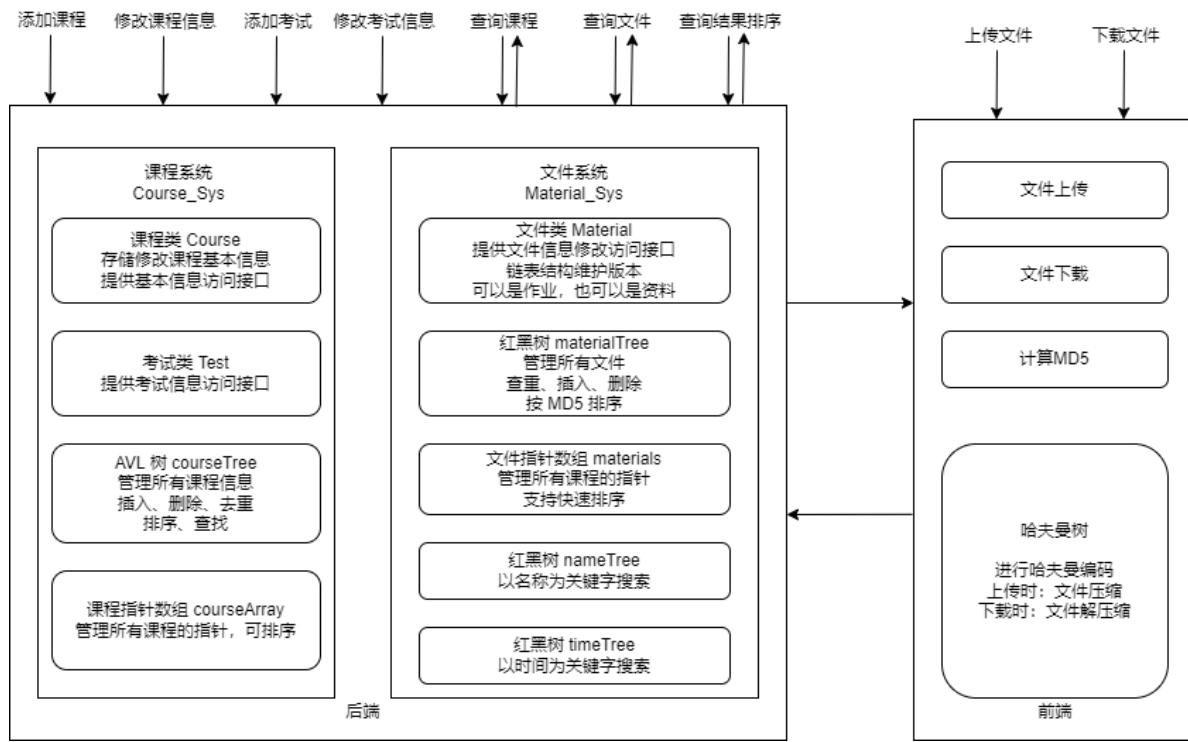
为实现上述功能，我们将整个课程管理系统模块进行拆分，拆分为课程系统和文件系统两部分，此外还有位于前端实现的文件上传下载相关模块。

课程系统包括两个基本类：课程类 Course，考试类 Test，和一个最外层的封装类：课程系统 CourseSys。课程系统需要支持插入、查询、避免重名等功能，我们采用一棵 AVL 平衡树来对所有课程进行维护。

文件系统包括一个基本类文件 Material，和一个封装类文件系统 MaterialSys。文件系统需要实现查重、去重、查找等功能，去重的依据是MD5算法得到的MD5码，我们采用一棵红黑树来进行维护这个值；高效查询则使用多棵不同关键字的红黑树实现。

前端的文件模块需要支持用户上传、下载，在前端加以实现；文件压缩采用哈夫曼树（哈夫曼编码）进行压缩和解压缩。

7.3 系统结构与各文件调用关系



7.4 基本算法

7.4.1 KMP 算法

算法描述

对于按照资料名称搜索文件，我们考虑到非完全匹配的情况，使用KMP算法来处理。具体而言，如果用户在搜索时输入了一个字符串 t ，我们支持搜索所有文件名含有子串 t 的文件，为了解决这个字符串匹配的问题，我们选择了效率非常好的KMP算法。

为了方便模块化开发，我们将KMP单独封装成了一个类，类定义和接口函数如下：

```
class KMP{
private:
    char* s;
    char* t;
    int lens, lent;
    bool exist;
    Vector<int> next;
    Vector<int> ans;
    void solve();

public:
    KMP(String ss, String tt);
    ~KMP();
    bool isExist();
    Vector<int> getPos();
};
```

类的成员变量中， s 为待匹配的字符串， t 为用来匹配的字符串， $lens$, $lent$ 分别为 s 和 t 的长度。 $next$ 数组即为用于 KMP 算法匹配的 $next$ 数组， ans 数组为记录匹配成功位置的数组。

KMP算法首先对匹配用的模式串进行一次自匹配，求出模式串各下标的最长前缀和最长后缀相同的长度，记录在 next 数组当中。当开始新一位的匹配时，要判断此时两个字符串（自匹配时，两字符串是同一个）是否仍匹配，若匹配，当前位置 + 1，否则，根据 next 数组往前跳转，再次进行匹配，直到跳转到开头或匹配成功为止。这部分在 KMP 类的构造函数中实现，具体代码如下：

```
KMP::KMP(String ss, String tt) { // 用 t 匹配 s
    s = ss.data();
    t = tt.data();
    lens = ss.getSize();
    lent = tt.getSize();
    int j = 0;
    next.pushBack(0);
    for (int i = 1; i <= lent; i++) next.pushBack(0);
    for (int i = 2; i <= lent; i++) {
        while (j && t[j] != t[i-1]) j = next[j];
        if (t[j] == t[i-1]) j++;
        next[i] = j;
    }
    exist = 0;
    solve();
}
```

在完成对于 next 数组的计算后，可以用和求 next 数组类似的方法进行匹配。具体来说，开始新一位匹配时，判断是否字符相同，若相同则匹配，位置后移一位；否则不匹配，那么模式串根据 next 数组向前跳转，再次判断是否匹配，直到跳转到开头或匹配成功为止。这部分在 KMP 类的 solve 函数中实现，具体代码如下：

```
void KMP::solve() {
    int j = 0;
    for (int i = 1; i <= lens; i++) {
        while (j && s[i - 1] != t[j]) j = next[j];
        if (j < lent && s[i - 1] == t[j]) j++;
        if (j == lent) {
            ans.pushBack(i - lent);
            j = next[j];
            exist = 1;
        }
    }
}
```

在此之外，KMP 类还封装了返回两字符串是否匹配的 bool `isExist()` 函数和返回所有匹配位置的 `Vector getPos()` 函数。代码比较简单，逻辑非常直观，在此略去，具体可以参看源代码部分。

时间复杂度分析

设待匹配串长 ls ，匹配串长 lt 。由 KMP 算法前述分析可知，在进行自匹配或者模式串匹配时，循环中被匹配串绝不回退。因此，计算 next 数组时，算法复杂度为 $O(lt)$ ；计算串匹配时，算法复杂度为 $O(ls)$ ，总复杂度为 $O(ls + lt)$ 。

需要注意的是，上述分析并不足够严格。虽然最终复杂度结果无误，但实际上，被匹配串每一位匹配的次数取决于目前的 next 数组，可能并不仅仅匹配一次，但我们依然可以证明，即使最坏情况下，KMP 复杂度也不会超过 $O(2(n+m))$

我们分析其总迭代次数一定存在一个上界 k ，而 $k < 2n$

对于 KMP 代码如下，令 $k = 2i - j$ ：

```

while ( j < m && i < n ) // 下面分析为何 k 一定为迭代次数上界
// 首先, k 必然是随着迭代单调递增的
if ( 0 > j || T[i] == P[j] ) // 匹配次数为 1
{ i ++; j ++; } // k 也恰好 + 1
else
j = next[j]; // 每次迭代, k 至少 + 1

```

算法结束时，一定有 $k = 2 * i - j \leq 2(n-1) - (-1) = 2n - 1 < 2n$

故 KMP 复杂度严格得证。

测试数据及结果

7.4.2 AVL 树

算法描述

AVL树是一种基本的自平衡二叉搜索树，其主要特点为，任何节点的左右子树高度差严格小于等于1。

在本项目中，使用 AVL 树维护所有的课程，比较关键字为课程的名称。由于二叉搜索树的特点，我们可以很方便地在树上按名称搜索课程，又因为它是一棵严格的平衡树，所以每一次插入、查找的时间复杂度都较优，不用担心二叉搜索树在动态插入的过程中出现退化的问题。

值得一提的是，由于我们同时实现了红黑树，所以此处 AVL 树的功能全部可以被红黑树替代，且红黑树拥有随机数据下更好的期望效率。但出于尽可能尝试更多样的数据结构的考虑，我们依然在维护课程时候保留了 AVL 树。

AVL 树的模板类定义：

```

template <class T>
class AVL {
private:
    int size;
    class AVLNode {
public:
    T data;
    int size, deep;
    AVLNode* left;
    AVLNode* right;
    AVLNode();
    explicit AVLNode(T dt);
    AVLNode(int s, int d, T dt);
    AVLNode(int s, int d, T dt, AVLNode* l, AVLNode* r);
    AVLNode(const AVLNode &other);
    ~AVLNode();
    int getSize(AVLNode *x);
    int getDeep(AVLNode *x);
    int getSize();
    int getDeep();
    AVLNode* ls();
}

```

```

    AVLNode* rs();
};

AVLNode* root;
void update(AVLNode *x);
void leftLeft(AVLNode** u);
void rightRight(AVLNode** u);
void leftRight(AVLNode** u);
void rightLeft(AVLNode** u);

public:
AVL();
~AVL();
AVLNode** getRoot();
void Free(AVLNode** u);
bool insert(AVLNode** u, T d);
bool exist(AVLNode* u, T d);
T search(AVLNode* u, T d);
bool Delete(AVLNode** u, T d);
void show_AVL(AVLNode* u);
};

```

在 AVL 类的内部，我们定义了一个子类 AVLNode，其含义是 AVL 树的节点类型。AVLNode 类的成员包括：存储的数据 data，所在子树的大小 size，所在子树的深度 deep，左子树指针 left，右子树指针 right。AVLNode 类的接口函数包括：getSize() 获得子树的大小，getDeep() 获取子树的深度，ls() 获得左子树指针，rs() 获得右子树指针。getSize() 和 getDeep() 根据调用方式可能有所不同进行了重载，实际功能没有太大区别。

在完成定义 AVLNode 类之后，我们可以定义好 AVL 树的所有成员变量：包括一个根节点指针 root 和树大小 size，其他节点通过节点间的指针关系来进行管理。

关键函数与接口介绍：

(1) 更新节点：void update(AVLNode* x)

更新指针 x 所指向的节点的 size 和 deep 值。节点 *x 的大小 size 为左右子树 size 相加再加上节点 *x 自身；节点 *x 的深度 deep 为左右子树 deep 的最大值再加 1.

代码实现为：

```

template <class T>
void AVL<T>::update(AVLNode *x) {
    if (x == NULL) return;
    x->size = x->getSize(x->left) + x->getSize(x->right) + 1;
    x->deep = Basic :: getMax<int>(x->getDeep(x->left),
                                         x->getDeep(x->right)) + 1;
}

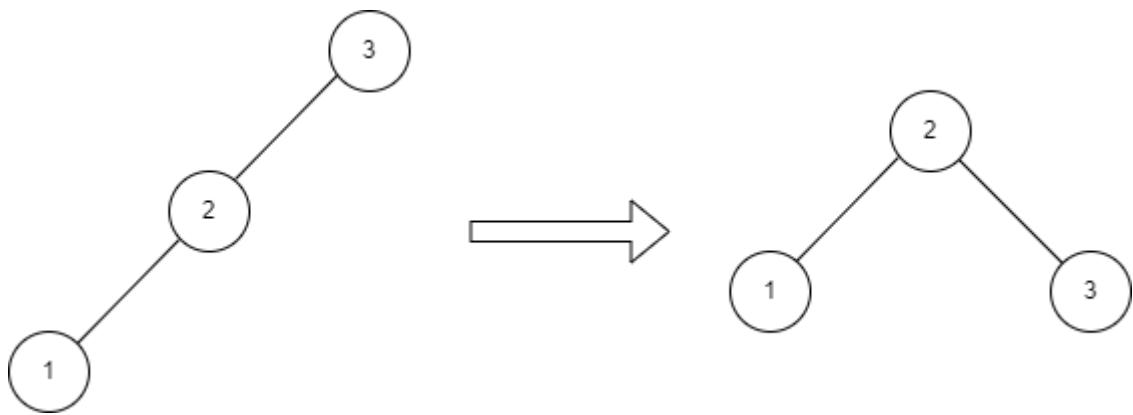
```

(2) 旋转操作

在插入或删除节点后，一棵原本平衡的二叉搜索树，可能会变得不平衡。为了实现树的自平衡，AVL 树通过旋转操作来保证自平衡。

二叉树不平衡的情况有如下四种：

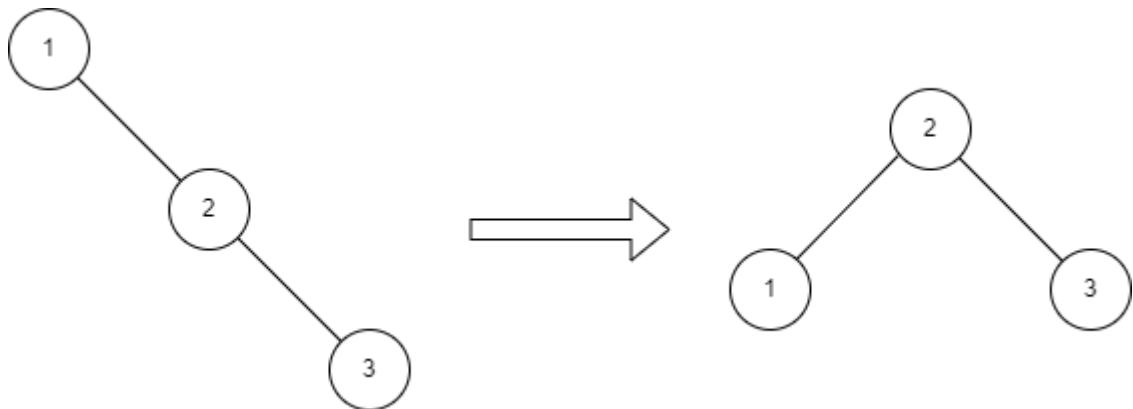
- 左左



右旋一次即可

```
template <class T>
void AVL<T>::leftLeft(AVLNode** u) {
    AVLNode* v = (*u) -> left;
    (*u) -> left = v -> right;
    v -> right = *u;
    update(*u);
    update(v);
    *u = v;
}
```

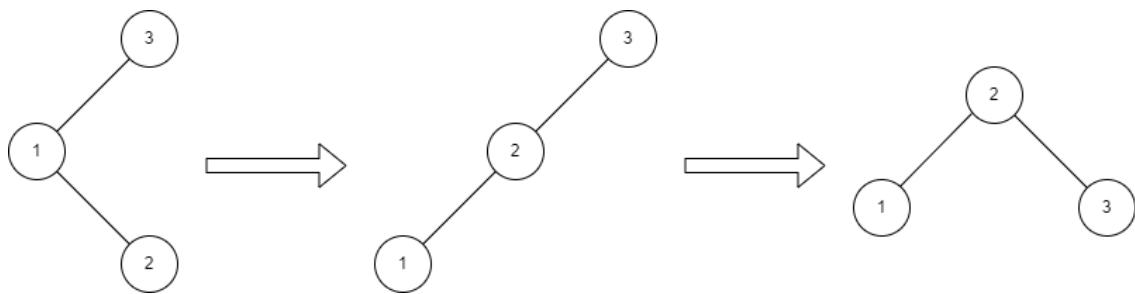
- 右右



左旋一次即可

```
template <class T>
void AVL<T>::rightRight(AVLNode** u) {
    AVLNode* v = (*u) -> right;
    (*u) -> right = v -> left;
    v -> left = *u;
    update(*u);
    update(v);
    *u = v;
}
```

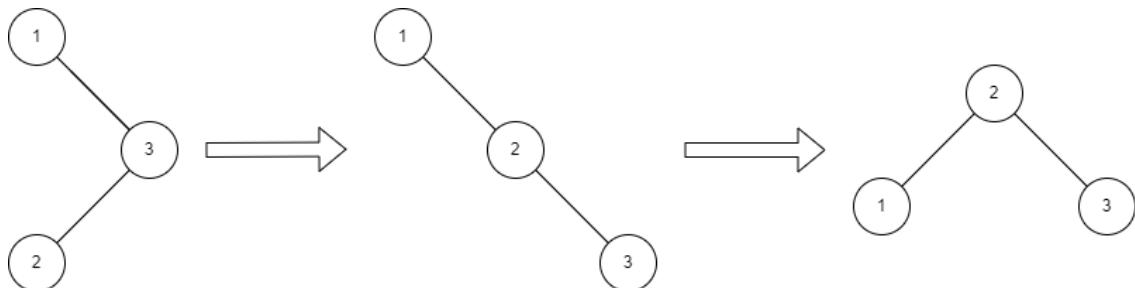
- 左右



左右的情况稍微复杂一些，要先进行一次右旋，将其转化为左左，再进行一次左旋

```
template <class T>
void AVL<T>::leftRight(AVLNode** u) {
    rightRight(&(*u) -> left);
    leftLeft(u);
}
```

- 右左



右左的情况类似左右，要先进行一次左旋，将其化为右右，再进行一次右旋

```
template <class T>
void AVL<T>::rightLeft(AVLNode** u) {
    leftLeft(&(*u) -> right);
    rightRight(u);
}
```

这些旋转操作保证了 AVL 树的自平衡特性。

(3) 插入函数：bool AVL :: insert(AVLNode ** u, T d)

在平衡树中插入数据 d，如果成功插入返回 0，如果已经存在了数据 d，那么不重复插入，返回 1。插入时，首先判断当前节点数据和待插入数据是否相等，若相等返回 1。若不相等，比较当前节点数据和待插入数据的大小关系，若大于，则递归在右子树插入此数据；若小于，则递归在左子树插入此数据。直到某节点为空，则将该节点替换为此数据项。

递归返回时，应判断左右子树的平衡性是否保持，若左右子树不平衡，那么根据上述分类的四种情形，进行相应的旋转操作，使子树恢复平衡。

代码实现为：

```
template <class T>
bool AVL<T>::insert(AVLNode** u, T d) {
    if (*u == NULL) {
        *u = new AVLNode(d);
        return 0;
    }
    if (d == (*u)->data) return 1;
```

```

if (d < (*u)->data) {
    bool f = insert(&((*u)->left), d);
    if (f) return 1;
    update(*u);
    if ((*u)->getDeep((*u)->left) - (*u)->getDeep((*u)->right) == 2) {
        if (d < (*u)->left->data) leftLeft(u);
        else
            leftRight(u);
    }
} else {
    bool f = insert(&((*u)->right), d);
    if (f) return 1;
    update(*u);
    if ((*u)->getDeep((*u)->right) - (*u)->getDeep((*u)->left) == 2) {
        if (d < (*u)->right->data) rightLeft(u);
        else
            rightRight(u);
    }
}
update(*u);
return 0;
}

```

(4) 判断存在函数: bool exist(AVLNode* u, T d)

判断 AVL 树中是否存在值为 d 的数据项。根据 AVL 树二叉搜索树的性质，只需要根据 d 在 AVL 树上搜索即可。首先判断 d 与当前节点的数据是否相等，若相等，则返回 1；若当前节点为空，则表示 AVL 树上不存在以 d 为数据的节点，查找失败，返回 1。

如果 d 比当前节点的数据小，那么递归查找当前节点的左子树；如果 d 比当前节点的数据大，那么递归查找当前节点的右子树。

代码实现为：

```

template <class T>
bool AVL<T>::exist(AVLNode* u, T d) {
    if (u == NULL) {
        return 0;
    }
    if (d == u->data) return 1;
    if (d < u->data) {
        bool f = exist(u->left, d);
        if (f) return 1;
    } else {
        bool f = exist(u->right, d);
        if (f) return 1;
    }
    return 0;
}

```

(5) 查找值函数: T search(AVLNode* u, T d)

在 AVL 树中查找与 d 相等的节点。如果找到返回对应节点的数据，否则返回 d 自身。具体递归思路和 exist 函数一致。

代码实现为：

```

template <class T>
T AVL<T>:: search(AVLNode* u, T d) {
    if (u == NULL) {
        return d;
    }
    if (d == u->data) return u->data;
    if (d < u->data) {
        T f = search(u->left, d);
        return f;
    } else {
        T f = search(u->right, d);
        return f;
    }
    return d;
}

```

(6) 删除函数: bool Delete(AVLNode** u, T d)

和大多数平衡树一样, AVL 树的删除较为复杂, 需要讨论删除后, 会出现什么情形的不平衡情况, 依次做旋转操作处理, 整个函数是采用递归的方式实现的。

如果 d 小于当前节点的数据, 那么在当前节点的左子树进行删除, 删除完成后, 有可能出现左左、左右两种不平衡情况, 判断是否不平衡并做出相应的旋转。

如果 d 大于当前节点的数据, 那么在当前节点的右子树进行删除, 删除完成后, 有可能出现右左、右右两种不平衡情况, 判断是否不平衡并做出相应的旋转。

如果 d 恰好需要删除当前的节点。首先判断当前节点的左右子树是否为空, 若有子树为空, 那么直接用另一个非空子树替代当前节点即可。否则在当前节点的右子树里面找到一个最小的元素, 将其赋值给当前节点, 再递归地在右子树中删除刚刚找到的最小元素即可。同样, 右子树删除后, 有可能出现右左、右右两种不平衡情况, 判断并进行相应的旋转即可。

代码实现为:

```

template <class T>
bool AVL<T>::Delete(AVLNode** u, T d) {
    if (*u == NULL) return 0;
    if (d < (*u)->data) {
        bool f = Delete(&((*u)->left), d);
        if (f == 0) return 0;
        update(*u);
        if ((*u)->getDeep((*u)->right) - (*u)->getDeep((*u)->left) == 2) {
            if ((*u)->getDeep((*u)->right->left) > \
                (*u)->getDeep((*u)->right->right))
                rightLeft(u);
            else
                rightRight(u);
        }
    } else {
        if (d > (*u)->data) {
            bool f = Delete(&((*u)->right), d);
            if (f == 0) return 0;
            update(*u);
            if ((*u)->getDeep((*u)->left) -
                (*u)->getDeep((*u)->right) == 2) {
                if ((*u)->getDeep((*u)->left->right) > \
                    (*u)->getDeep((*u)->left->left))
                    leftRight(u);
            }
        }
    }
}

```

```

        else
            leftLeft(u);
    }
} else {
    if ((*u)->left == NULL || (*u)->right == NULL) {
        if ((*u)->left) {
            *u = (*u)->left;
            return 1;
        }
        if ((*u)->right) {
            *u = (*u)->right;
            return 1;
        }
        *u = NULL;
        update(*u);
        return 1;
    } else {
        AVLNode* v = (*u)->right;
        while (v->left) v = v->left;
        (*u)->data = v->data;
        Delete(&(*u)->right), v->data);
        update(*u);
        if ((*u)->getDeep((*u)->left) -
            (*u)->getDeep((*u)->right) == 2) {
            if ((*u)->getDeep((*u)->left->right) > \
                (*u)->getDeep((*u)->left->left))
                leftRight(u);
            else
                leftLeft(u);
        }
    }
}
update(*u);
return 1;
}

```

因为根据代码规范要求不允许直接传引用，所以指针操作写得有点复杂。

时间复杂度分析

当根据 BST 规则进行操作时，AVL 的平衡性可能被破坏，显然，这种平衡性的破坏仅涉及到该节点的祖先。对于插入操作，当插入一个节点后，从祖父开始，该节点的每个祖先都有可能失衡，且可能不止一个；对于删除操作，当删除一个节点后，从父亲开始，每个祖先都可能失衡，但只可能有一个失衡。为了解决失衡问题，必须进行旋转操作。

由前述分析可知，AVL 的所有旋转操作都在局部进行，只涉及几个节点的指针交换。因此，每一次旋转复杂度都为 $O(1)$ ，由之前分析可知，在每一深度只需要检测并旋转至多 1 次。故总复杂度为 $O(\log n)$

值得一提的是，虽然删除操作时，AVL 只有至多一个节点失衡，但是其依然需要 $O(\log n)$ 的复杂度，这是由于 AVL 在调整失衡节点时，可能造成更高祖先的失衡，失衡会从第一个不平衡节点开始向上传递，最多造成 $\log n$ 个节点先后失衡。故删除复杂度 $O(\log n)$

模块测试用例及结果

对于 AVL 树模块，我们以整型数据为例进行了测试，测试代码如下：

```
#include "AVL.cpp"
#include <bits/stdc++.h>
using namespace std;

AVL<int> avl;

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        avl.insert(avl.getRoot(), x);
    }
    avl.show_AVL(*avl.getRoot());
    cout << endl;
    avl.show_AVL_2(*avl.getRoot());
    cout << endl;
    for (int i = 0; i < q; i++) {
        int x;
        cin >> x;
        if (x == 1) {
            int y;
            cin >> y;
            avl.insert(avl.getRoot(), y);
        } else if (x == 2) {
            int y;
            cin >> y;
            avl.Delete(avl.getRoot(), y);
        } else if (x == 3) {
            int y;
            cin >> y;
            cout << (avl.exist(*avl.getRoot(), y) ? "Yes" : "No") << endl;
        } else if (x == 4) {
            avl.show_AVL(*avl.getRoot());
            cout << endl;
            avl.show_AVL_2(*avl.getRoot());
            cout << endl;
        }
    }
    system("pause");
}
```

测试用的数据如下：

```
7 8  
25 43 78 1 16 -2 0  
3 25  
2 25  
3 25  
4  
3 15  
1 15  
3 15  
4
```

测试结果如下：

运行结果如下图。测试时，首先插入了 7 个乱序的整数，插入到AVL树中后，输出AVL树中序遍历结果，输出的结果有序。随后进行 8 次操作，第一次询问 25 是否在AVL树中，由于之前已经插入，得到结果 Yes。第二次操作删去 25，第三次操作询问 25 是否在AVL树中，由于已删除，得到结果 No。第四次操作输出 AVL 树的中序遍历结果，共 6 个数据，结果有序。第五次操作询问 15 是否在AVL树中，此时15 尚未插入，得到结果 No。第六次操作将 15 插入到AVL树中，第七次操作询问 15 是否在红黑树中，得到结果 Yes。最后输出AVL树中序遍历结果，共 7 个数据，结果有序。

```
E:\Project\CourseAuxiliarySystem\src\test.exe  
7 8  
25 43 78 1 16 -2 0  
0 -2 1 16 43 25 78  
3 25  
Yes  
2 25  
3 25  
No  
4  
0 -2 1 16 43 78  
3 15  
No  
1 15  
3 15  
Yes  
4  
0 -2 1 15 16 43 78  
请按任意键继续. . .
```

7.4.3 MD5算法

对于资料和作业的上传下载，我们需要实现文件的去重功能。朴素的做法是直接遍历整个文件，但这样的做法不仅效率低下，而且会占用大量内存空间。为了节省空间，提高效率，我们采用 MD5 算法，先求出文件的散列哈希值作为特征值，再按照这一特征值去重。MD5信息摘要算法是一种散列算法，由于运行稳定、计算速度快、安全性较好等特点被广泛使用。由于 MD5 算法产生的哈希值长度固定（128-bit），文件去重的空间复杂度即为 $O(\text{文件数量})$ ，大幅减少了内存开支。

算法描述

- 一些函数和常数

MD5 算法引入了一些函数和常量用于处理数据。首先是四个非线性函数，定义如下：

```

F(x, y, z) = (x & y) | ((~x) & z)
G(x, y, z) = (x & z) | (y & (~z))
H(x, y, z) = x ^ y ^ z
I(x, y, z) = y ^ (x | (~z))

```

然后是一个常量数组 $\{t_i\}$:

$$t[i] = \lfloor 2^{32} \cdot |\sin(i+1)| \rfloor \quad i \in [0, 64) \cup \mathbb{Z}$$

- 填充数据

MD5 算法首先将原始数据进行填充，使其长度为 512 的倍数。具体步骤如下：

- 首先，在原始数据末尾补一个 1
- 然后，在末尾补若干个 0，直至数据长度模 512 恰为 448
- 最后，设原始数据长度为 len ，将 len 的低 64 位填充到数据末尾，最终数据长度为 512 的倍数

注意无论原始数据长度是多少，都必须先补一个 1。

- 分组处理

数据填充完成后，MD5 算法将数据以 512 位为一组进行循环处理，记这 512-bit 的数据为 M 。MD5 算法用四个 32 位的寄存器 A, B, C, D 保存散列结果，最后将 A, B, C, D 顺次连接即为结果，其初始值为：

$$\begin{aligned} A &= 0x01234567 \\ B &= 0x89abcdef \\ C &= 0xfedcba98 \\ D &= 0x76543210 \end{aligned}$$

每次循环中，MD5 算法对 M, A, B, C, D 进行处理得到新的 A, B, C, D ，进行下一次循环。

- 再次分组

每次循环中。我们将 M 再次分为 16 组，每组 32 位，记为 $\{m_i\}$ ，同时，令 a, b, c, d 为 A, B, C, D 的拷贝。

接下来，我们进行四轮处理，方便起见，用伪代码进行描述：

- 预处理

```

slice M to m[i] i = 0 to 15
a <- A, b <- B, c <- C, d <- D

```

- 第一轮

```

s1[] = {7, 12, 17, 22}
for index = 0 to 15
    i <- index
    j <- index
    k <- index % 4
    a <- b + ((a + F(b, c, d) + m[i] + t[j]) << s1[k])
    {a,b,c,d} <- {d,a,b,c}

```

循环内的最后一行表示将 $\{a, b, c, d\}$ 向右轮换。此处的左移表示循环左移，即 ROL，下文同理。

- 第二轮

```

s2[] = {5,9,14,20}
for index = 0 to 15
    i <- (index * 5 + 1) % 16
    j <- index + 16
    k <- index % 4
    a <- b + ((a + G(b, c, d) + m[i] + t[j]) << s2[k])
    {a,b,c,d} <- {d,a,b,c}

```

- 第三轮

```

s3[] = {4,11,16,23}
for index = 0 to 15
    i <- (index * 3 + 5) % 16
    j <- index + 32
    k <- index % 4
    a <- b + ((a + H(b, c, d) + m[i] + t[j]) << s3[k])
    {a,b,c,d} <- {d,a,b,c}

```

- 第四轮

```

s4[] = {6,10,15,21}
for index = 0 to 15
    i <- (index * 7) % 16
    j <- index + 48
    k <- index % 4
    a <- b + ((a + I(b, c, d) + m[i] + t[j]) << s4[k])
    {a,b,c,d} <- {d,a,b,c}

```

- 累加

在这四轮处理完成后，我们得到了新的 a, b, c, d 。随后，我们将其累加到 A, B, C, D 上，并进行下一次循环：

```

A <- A + a
B <- B + b
C <- C + c
D <- D + d

```

- 统计答案

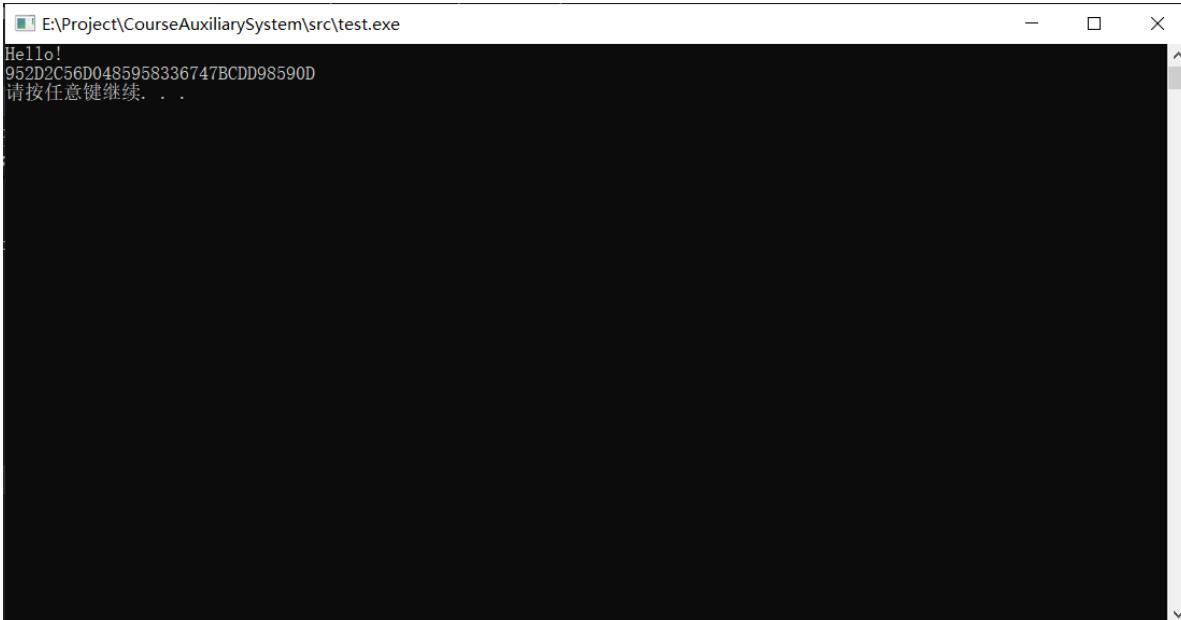
正如之前所说，将 A, B, C, D 顺次连接即为原信息的 MD5 码。

模块测试数据及结果

测试程序如下：

```
int main() {
    char s[233];
    scanf("%s", s);
    int n = strlen(s);
    ByteVector S;
    for (int i = 0; i < n; i++)
        S.pushBack(s[i]);
    ByteArray T = MD5 :: calculateMD5Code(S);
    for (int i = 0; i < T.getSize(); i++)
        printf("%.2X", T[i]);
    return 0;
}
```

运行结果如下：



验证无误。

7.4.4 红黑树

在求出每个文件的信息后，我们以 MD5 码为关键字，用红黑树进行维护，来实现文件的去重。红黑树是一种效率较高的自平衡二叉查找树，广泛用于Linux 的进程管理、内存管理，设备驱动及虚拟内存跟踪等一系列工程实践中。核心思想是在二叉查找树中引入了节点颜色的概念，并规定了一系列性质，在插入删除时维护这些性质，达到高效查找、维护的目的，统计性能优于 AVL 树等数据结构。

红黑树的定义

红黑树是一棵二叉搜索树，满足二叉搜索树的所有性质，在此基础上，红黑树每个节点都有一个颜色属性，并满足如下性质：

- 节点的颜色为红色或黑色
- 根节点是黑色
- 不能有两个连续的红色节点（即红色节点的孩子必须是黑色）
- 从任一节点到其子树内每个叶子的路径上的黑色节点数量相同

类声明如下：

```

enum RBTCOLOR {RED, BLACK};

template<class T>
struct RBTNode {
    T data;
    RBTCOLOR color;
    RBTNode *parent;
    RBTNode *left, *right;
    RBTNode();
    RBTNode(T value, RBTCOLOR c, RBTNode *p, RBTNode *l, RBTNode *r);
};

template <class T>
struct RBTree {
private:
    RBTNode<T> *root;
    void leftRotate(RBTNode<T> *x);
    void rightRotate(RBTNode<T> *x);
    void insertFixUp(RBTNode<T> *x);

public:
    RBTree();
    ~RBTree();

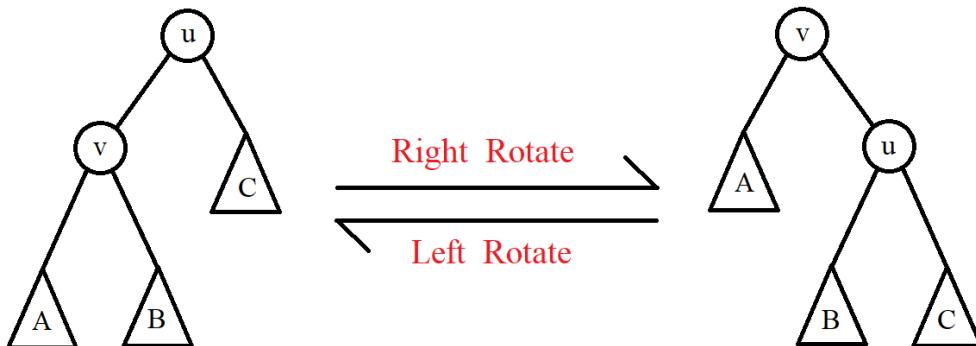
    void insert(T data);
    bool find(T key);
};

```

在这里，我们定义了枚举类 RBTCOLOR 表示节点颜色类，该类有两个取值，RED 和 BLACK。这样做可以有效防止非法情况出现。RBTNode 为红黑树的节点类，这里采用模板类方便复用。其属性除了节点数据 data 和颜色 color 外，还有三个指针 parent, left, right，分别指向当前节点在红黑树上的父亲以及左右孩子。

红黑树的旋转

红黑树的旋转与其他带旋自平衡二叉搜索树类似，分为左旋与右旋，如图所示：



平衡树的旋转操作有一个重要的性质就是，旋转前后树的形态发生了变化，但是仍然满足原来二叉搜索树的性质。以右旋为例，上方左图对节点 u 进行右旋操作后，u 的左孩子 v 变为了 u 的父亲，而 v 的右子树挂到了 u 的左子树的位置上，树的形态发生了改变，但仍然满足二叉搜索树的性质。红黑树左旋右旋的代码如下：

```

template<class T>
void RBTree<T>::leftRotate(RBTNode<T> *x) {
    RBTNode<T> *y = x->right;
    x->right = y->left;
    if (y->left)
        y->left->parent = x;
    y->left = x;
    y->parent = x->parent;
    if (x->parent) {
        if (x->parent->left == x)
            x->parent->left = y;
        else
            x->parent->right = y;
        x->parent = y;
    } else {
        root = y;
        x->parent = y;
    }
}

template<class T>
void RBTree<T>::rightRotate(RBTNode<T> *x) {
    RBTNode<T> *y = x->left;
    x->left = y->right;
    if (y->right)
        y->right->parent = x;
    y->right = x;
    y->parent = x->parent;
    if (x->parent) {
        if (x->parent->left == x)
            x->parent->left = y;
        else
            x->parent->right = y;
        x->parent = y;
    } else {
        root = y;
        x->parent = y;
    }
}

```

红黑树的插入

将一个节点插入红黑树中，首先需要在二叉搜索树上找到需要插入的位置，将节点插入。随后，将节点颜色设为红色，来保证性质 4，随后，我们需要通过一系列的调整使其重新成为一棵红黑树。代码如下：

```

template<class T>
void RBTree<T>::insert(T data) {
    RBTNode<T> *node = NULL;
    if (!(node = new RBTNode<T>(data, RBTCOLOR::BLACK, NULL, NULL, NULL)))
        return;
    RBTNode<T> *v = NULL;
    RBTNode<T> *u = root;
    while (u) {
        v = u;
        if (node->data < u->data)
            u = u->left;
        else
            u = u->right;
    }
    if (v->parent)
        if (v->parent->left == v)
            v->parent->left = node;
        else
            v->parent->right = node;
    else
        root = node;
    node->parent = v;
    node->color = RBTCOLOR::RED;
    fix(node);
}

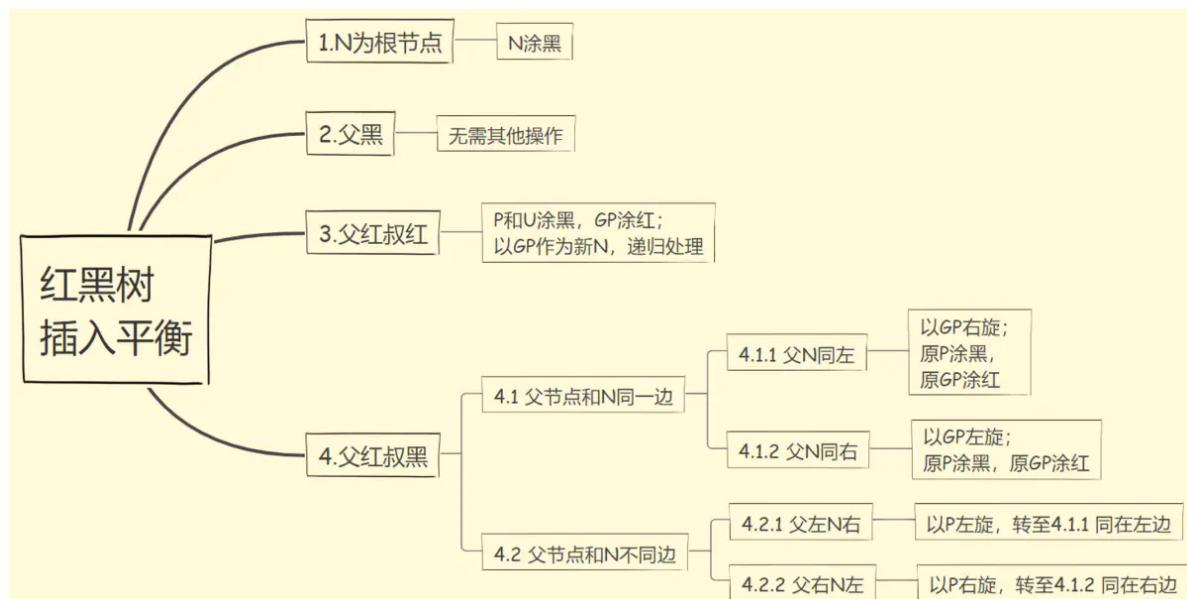
```

```

        else
            u = u -> right;
    }
    if (v) {
        node -> parent = v;
        if (node -> data < v -> data)
            v -> left = node;
        else
            v -> right = node;
    } else {
        node -> parent = v;
        root = node;
    }
    node -> color = RBTColor::RED;
    insertFixUp(node);
}

```

考虑直接插入后，红黑树的哪些性质会被打破。首先，性质1仍然满足，性质4由于新插入节点为红色，也依然满足。会被打破的只有性质2与性质3。若性质2被打破，则当前节点为根节点，直接改为黑色即可。若性质3被打破，则当前节点的父亲也是红色。接下来如下图分情况处理，直至所有性质都被满足。



代码如下：

```

template<class T>
void RBTree<T>::insertFixUp(RBTNode<T> *node) {
    RBTNode<T> *parent;
    RBTNode<T> *grandParent;
    while ((parent = node -> parent) && parent -> color == RBTColor::RED) {
        grandParent = parent -> parent;
        if (parent == grandParent -> left) {
            RBTNode<T> *uncle = grandParent -> right;
            if (uncle && uncle -> color == RBTColor::RED) {
                uncle -> color = RBTColor::BLACK;
                parent -> color = RBTColor::BLACK;
                grandParent -> color = RBTColor::RED;
                node = grandParent;
                continue;
            }
        }
    }
}

```

```

        if (parent -> right == node) {
            RBTNode<T> *tmp;
            leftRotate(parent);
            tmp = parent;
            parent = node;
            node = tmp;
        }

        parent -> color = RBTCOLOR::BLACK;
        grandParent -> color = RBTCOLOR::RED;
        rightRotate(grandParent);
    } else {
        RBTNode<T> *uncle = grandParent -> left;
        if (uncle && uncle -> color == RBTCOLOR::RED) {
            uncle -> color = RBTCOLOR::BLACK;
            parent -> color = RBTCOLOR::BLACK;
            grandParent -> color = RBTCOLOR::RED;
            node = grandParent;
            continue;
        }

        if (parent -> left == node) {
            RBTNode<T> *tmp;
            rightRotate(parent);
            tmp = parent;
            parent = node;
            node = tmp;
        }

        parent -> color = RBTCOLOR::BLACK;
        grandParent -> color = RBTCOLOR::RED;
        leftRotate(grandParent);
    }
}

root -> color = RBTCOLOR::BLACK;
}

```

红黑树的查找

为了处理文件去重，我们还需要在红黑树上查询某个文件，此时需要利用二叉搜索树的性质，这一部分在 AVL 树一节中已有详细描述，在此不多赘述，代码如下：

```

template<class T>
bool RBTree<T>::find(T key) {
    RBTNode<T> *u = root;
    while (u) {
        if (key == u -> data)
            return true;
        if (key < u -> data)
            u = u -> left;
        else
            u = u -> right;
    }
    return false;
}

```

时间复杂度分析

红黑树每次插入并调整后，都满足我们之前所说的性质。不妨设深度最浅的叶子（空节点）深度为 x ，深度最深的叶子深度为 y ，由性质4，根到两个叶子的路径上的黑色节点数量相同，又由性质三，红色点不能连续出现，因此 $y \leq 2x$ 。设红黑树总节点数为 n ，则有 $2^x - 1 < n \leq 2^y - 1$ ，可得 $y \leq 2 \log n$ 。因此，红黑树高度不超过 $2 \log n$ ，单次插入、查询的时间复杂度均为 $O(\log n)$

模块测试用例及结果

对于红黑树模块，我们以整型数据为例进行了测试，测试代码如下：

```
#include "rbtree.hpp"
using namespace std;

RBTree<int> A;

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 1, x; i <= n; i++) {
        cin >> x;
        A.insert(x);
    }
    A.print();
    for (int op, x; q--;) {
        cin >> op;
        if (op == 0) {
            cin >> x;
            A.insert(x);
        } else if (op == 1) {
            cin >> x;
            cout << (A.find(x) ? "Yes" : "No") << endl;
        } else {
            cout << "Current Rbtree: ";
            A.print();
            cout << endl;
        }
    }
}
```

测试用的数据如下：

```
7 5
25 43 78 1 16 -2 0
1 25
1 15
0 15
1 15
2
```

运行结果如下图。测试时，首先插入了7个乱序的整数，插入到红黑树中后，输出红黑树中序遍历结果，输出的结果有序。随后进行5次操作，第一次询问25是否在红黑树中，由于之前已经插入，得到结果Yes。第二次询问15是否在红黑树中，此时15尚未插入，得到结果No。第三次操作将15插入到红黑树中，第四次操作询问15是否在红黑树中，得到结果Yes。最后输出红黑树中序遍历结果，为-2 0 1 15 16 25 43 78，共8个数据，结果有序。

```
E:\Project\datastructure\rbtree\test.exe
7 5
25 43 78 1 16 -2 0
-2 0 1 16 25 43 78
1 25
Yes
1 15
No
0 15
1 15
Yes
2
Current Rbtree: -2 0 1 15 16 25 43 78
请按任意键继续. . .
```

7.4.5 哈夫曼树 Huffmantree

如果要实现文件的压缩，就需要通过哈夫曼树进行压缩和解压。

首先需要定义树节点以及节点的比较方式

```
class TreeNode {
public:
    unsigned char c;
    int weight;
    TreeNode* left;
    TreeNode* right;
    TreeNode();
    explicit TreeNode(int);
    TreeNode(unsigned char, int);
    ~TreeNode();
};

struct NodeCompare {
    bool operator()(TreeNode* lhs, TreeNode* rhs) const {
        return lhs->weight > rhs->weight;
    }
};
```

可以看到树节点的比较大小方式是通过比较节点权重大小进行。树节点中记录了节点代表的字符c以及该字符的权重，以及左右结点的指针。

成员函数有构造参数，可以通过传入权重以及字符的方式初始化树节点。

其次完成哈夫曼树的定义：

```
class HuffmanTree {
private:
    int64_t totalBits;
    int frequency[256];
    TreeNode* root;
```

```

    Heap<TreeNode*, NodeCompare> priqueue;
    std::string charCode[256];
    void inorder(TreeNode*, std::string);
    Vector<unsigned char> encode(const Vector<unsigned char>&);
    Vector<unsigned char> decode(const Vector<unsigned char>&);
    void bulidTree(int[], int);
    int64_t getTotalBits() const;
    void clear();
}

public:
    HuffmanTree();
    ~HuffmanTree();
    bool upload(const std::string, const std::string);
    bool download(const std::string, const std::string, const std::string);
};

```

在哈夫曼树中，totalbits表示压缩后字符数，frequency是每个字符出现的频率数组，priqueue是通过对实现的优先队列，在建树过程中使用，charCode用于存储字符编码后的01字符串。

成员函数包括：inorder中序遍历哈夫曼树以获取每个字符的编码方案，encode，传入待编码的字符串进行编码，decode，传入待解码的字符串进行解码，bulidTree，传入字符频率数组以及字符数量（默认为256）建立哈夫曼树，clear则是清空哈夫曼树内容。

接口函数为upload上传文件，传入原地址以及目的地址，是对于encode的封装，download下载文件，传入原地址，目的地址，文件名，是对于decode的封装。

算法描述

接下来重点讲解核心函数bulidTree,encode,decode。

buildTree中，先为权重不为0的字符创建树节点并放入堆中，随后当堆的大小大于1时，连续两次取出堆顶元素并创建这两个节点的根节点，根节点权重为两个节点之和，并将新节点压入堆中。

```

void HuffmanTree::bulidTree(int charweight[], int n = 256) {
    for (int i = 0; i < n; i++) {
        if (charweight[i] > 0) {
            TreeNode* node = new TreeNode((unsigned char)i, charweight[i]);
            priQueue.push(node);
        }
    }
    while (priQueue.getSize() > 1) {
        TreeNode* minNode1 = priQueue.top();
        priQueue.pop();
        TreeNode* minNode2 = priQueue.top();
        priQueue.pop();
        TreeNode* mergeNode = new TreeNode(minNode1->weight + minNode2->weight);
        mergeNode->left = minNode1;
        mergeNode->right = minNode2;
        priQueue.push(mergeNode);
    }
    root = priQueue.top();
    priQueue.pop();
}

```

encode中，我们先遍历待编码字符串获取每个字符出现的频率，随后建立哈夫曼树，并通过中序遍历获取字符编码方案，随后则是遍历待编码字符串——进行编码，同时记录字符串的总比特数（当编码至末尾时，如果总比特数不为8的整倍数，则需要补上0，因此需要记录总比特数用于解码时判定真正的结尾在哪里）。

```
vector<unsigned char> HuffmanTree::encode(const vector<unsigned char>& text) {
    Vector<unsigned char> result;
    int bitCount = 0;
    unsigned char temp = 0;
    std::string emptyString = "";
    for (int i = 0; i < text.getSize(); i++) {
        frequency[text[i]]++;
    }
    buildTree(frequency);
    inorder(root, emptyString);
    for (int i = 0; i < text.getSize(); i++) {
        std::string temps = charCode[text[i]];
        int length = charCode[text[i]].size();
        for (int j = 0; j < length; j++) {
            if (temps[j] == '0')
                temp = temp << 1;
            else
                temp = temp << 1 | 1;
            ++bitCount;
            ++totalBits;
            if (bitCount == 8) {
                result.pushBack(temp);
                temp = 0;
                bitCount = 0;
            }
        }
    }
    if (bitCount > 0) temp = temp << (8 - bitCount);
    result.pushBack(temp);
    return result;
}
```

在decode中，则是根据已有的哈夫曼树依次遍历比特串，当遇到树叶节点（左右节点为空）时证明完成一个字符的解码，将该字符放入结果数组中，直到totalbits归0证明解压完毕。

```
Vector<unsigned char> HuffmanTree::decode(const Vector<unsigned char>& code) {
    Vector<unsigned char> text;
    int count = 7;
    int index = 0;
    TreeNode* cur = root;
    // printf("%d\n",code.getSize());
    while (index < code.getSize() && totalBits > 0) {
        int bit = (code[index] & (1 << count)) == 0 ? 0 : 1;
        if (count == 0) {
            ++index;
            count = 8;
        }
        if (bit)
            cur = cur->right;
        else
            cur = cur->left;
    }
}
```

```

    if (cur->right == cur->left) {
        text.pushBack(cur->c);
        cur = root;
    }
    --count;
    --totalBits;
}
return text;
}

```

时空复杂度分析

1. 编码阶段：

建树主要分为三大部分：（1）建立字符出现次数映射表（2）通过哈希表建立优先队列（3）通过优先队列建哈夫曼树

首先，建立映射表时需要遍历输入字符串，设输入字符串的长度为 N ，因此这部分的时间复杂度为 $O(N)$

其次，设哈希表的元素数量为 n ，也就是字符串中包含的不重复字符数为 n ，在建立了优先队列时，堆的单次插入时间复杂度为 $O(\log n)$ ，因此总的时间复杂度为 $O(n \log n)$

最后，优先队列的元素数量与字符串中包含不重复的字符数相同，也为 n ，在循环中，我们每次取出两个元素并放入一个元素，也即是说每经历一次循环优先队列的大小都会减小1，直到队列中只剩一个元素，因此我们一共进行了 $n-1$ 次循环，时间复杂度为 $O(n)$

综上所述，建立一个哈夫曼树的总时间复杂度为 $O(N + n \log n)$ ，其中 N 为输入字符串的长度， n 为字符串中不重复的字符数量，而空间复杂度为 $O(n)$ ，因为建立了 n 个树节点同时还有映射表数组使用的空间

随后则是逐字符编码，该过程的时间复杂度为 $O(m)$ ， m 为编码字符串长度

2. 解码阶段：

由于我们需要遍历整个字符串，串中每个字符（0或1）都进行了一次递归，而递归函数中也没有循环，因此这一阶段的时间复杂度为 $O(N)$ ， N 为输入的待解码字符串的长度，而空间复杂度则是由于我们的递归函数有着系统栈开销，因此总的空间复杂度也为 $O(N)$

模块测试

哈夫曼树的测试详见前端文件系统部分。

7.4.6 资料文件类 Material

对于作业和课程资料，它们的管理和维护有一定的共性，因此我们将其抽象成一个资料文件类，基本定义如下：

```

class Material {
private:
    bool homework;
    String courseName;
    String name;
    ByteArray md5;
    Time updateTime;
    String path;
    Material* nextVersion = NULL;
}

```

```

public:
    Material();
    Material(bool h, String name, \
             ByteArray m, String n, String p, Time u);
    ~Material();
    explicit Material(const Material& other);
    bool isHomework();
    bool isData();
    String getCourseName();
    String getName();
    ByteArray getMd5();
    Time getTime();
    String getPath();
    Material* getLatestVersion();
    void setNextVersion(Material* m);
    Material* getNextVersion();
    Material& operator = (const Material& other);
};


```

其成员变量分别表示：homework，记录是否是家庭作业，若是家庭作业，homework = 1；若是课程资料，homework = 0。courseName 记录对应的课程名称。name 记录对应的作业项或资料项的名称，若两个文件拥有相同的 name，那么将它们视为同一个资料或作业的不同版本。md5 记录上传至系统的文件的 md5 码，我们用 md5 码进行判重。updateTime 记录上传的时间。path 记录该文件的路径。nextVersion 用于版本管理，记录同一个资料或作业，下一个版本的指针。

其接口函数包括：构造函数，允许传入多个参数构造一个 Material 类；isHomework() 函数，判断是否为家庭作业；isData() 函数，判断是否是课程资料；getCourseName() 函数，返回此课程的名字；getName() 函数，返回此作业项的名字；getMd5() 函数，返回此资料的 md5 码；getTime() 函数，返回此资料的更新时间；getPath() 函数，返回此资料对应的文件路径；getLatestVersion() 函数，返回此资料的最新版本；setNextVersion() 函数，为当前资料设置下一个版本；getNextVersion() 函数，获得当前资料的下一个版本。

下面具体分析一下版本控制相关的部分：资料类型的版本控制是通过一个链表来进行的，每次上传一个资料项的新版本，系统会自动判断这个资料项此前是否已经上传过，若没有上传过，直接新增一个需要维护的资料项；如果上传过了，系统会在原本资料项的链表后面新插入刚刚上传的资料。因此，我们过往上传的版本都通过这个链表维护起来，可以支持获得最新版本、获得下一个版本、查询过往版本的操作，具体相关的几个函数代码如下：

```

Material* Material::getLatestVersion() {
    Material* latest = this;
    while (latest->nextVersion != NULL) {
        latest = latest->nextVersion;
    }
    return latest;
}
void Material::setNextVersion(Material* m) {
    nextVersion = m;
}
Material* Material::getNextVersion() {
    return nextVersion;
}

```

其他接口函数比较简单，在此略去。

7.4.7 资料管理系统类 MaterialSys

资料管理系统类 MaterialSys 是对于文件资料管理这部分最外层的类，和 CourseSys 类一同，作为课程管理模块最外层的接口。主要实现了导入资料到系统、按照资料项名称和上传时间获得资料、按照资料名称进行关键字搜索、获得特定版本的资料、按照名称和时间对资料进行排序输出等。

上传、压缩和下载虽然在后端留有基本的代码，主要功能依然是在前端实现，因此不在这部分进行描述。

其类定义部分代码为：

```
class Materialsys {
private:
    Vector<Material*> materials;
    RBTree<MaterialPtr> materialTree;
    RBTree<Spair<String, int>> nameTree;
    RBTree<Spair<Time, IntPair>> timeTree;
    Vector<Material*> mForSort;

public:
    Materialsys();
    ~Materialsys();
    bool addMaterials(Material* m);
    Material* getMaterialByName(const String& name);
    Material* getMaterialByTime(const Time& t);
    Vector<Material*> search(const String& name);
    Material* getCertainVersion(const String& name, int t);
    void sortByTime(int l, int r);
    void sortByName(int l, int r);
    Vector<Material*> getAllMaterial();
}
```

下面对关键功能进行介绍：

(1) 红黑树实现文件判重

我们依据文件的MD5码进行判重，如果两个文件的MD5码一致，我们就认为这两个文件是重复文件。我们用一棵以MD5码为比较关键字的红黑树来维护所有的文件。

```
RBTree<MaterialPtr> materialTree;
```

这里的 MaterialPtr 是一个重载了 < 和 = 的 Material 类指针，重载运算符后，能够根据MD5码的大小关系进行比较。

上传文件时，首先在 materialTree 中调用函数 materialTree.find(x) 判断当前传入的文件是否重复，如果重复，返回 false，表示上传失败；如果不重复，将其作为一个新节点，插入 materialTree 当中。由于上传文件还涉及其他的几个功能，函数的具体思路后续再进一步说明。

(2) 红黑树实现快速查找

红黑树是一棵自平衡的二叉搜索树，因此，在红黑树上根据关键字查找效率非常高，复杂度为严格的 $O(\log N)$ 。根据红黑树的这一优势，我们可以实现文件资料的快速查找。

为了方便调用，所有的红黑树查找均返回一个整型 int 或一个整形对 IntPair，表示所查询的文件资料在 materials 数组中的下标，和对应的版本号。为了实现便捷查询，我们仿照 Pair 类额外定义了 Spair 类，与 Pair 类不同，Spair 比较时仅比较第一关键字 first，如果第一关键字相同，就认为两个 Spair 相同。Spair 可以作为红黑树的节点数据类型，first 为需要查询的关键字，second 为查询返回的

答案。这样，当查找到红黑树上的特定节点时，将查询的关键字和答案一起作为红黑树的数据类型返回出来。

```
RBTree<Spair<String, int>> nameTree;
RBTree<Spair<Time, IntPair>> timeTree;
```

在每次新加入文件资料时，首先在 nameTree 当中通过 nameTree.find(x) 查询该资料是否已经出现过，如果已经出现过，那么现在上传的是同一个资料项的新版本，那么通过 nameTree.search(x) 获得对应资料项在 materials 数组中的下标，即获得该资料项的头，在链表最后加入新资料，并记录在 materials 数组中的下标和版本编号。根据记录的下标和版本编号，构建一个 Spair<Time, IntPair>，插入查询时间的红黑树 timeTree 中。因为当前资料项是已经出现过的，因此不需要更新 nameTree。如果该资料从未出现过，那么在更新上述 timeTree 之外，还需要额外更新 nameTree。具体为，在变长数组 materials 最后 push_back 新资料，记录其下标，构建一个 Spair<String, int>，其 first 为文件资料名称，second 为下标，将其插入 nameTree 中。

在此时，我们已经完整描述了新插入一个文件资料的算法过程，这里给出外部接口函数 bool addMaterials(Materials* m) 的具体实现：

```
bool MaterialsSys::addMaterials(Material* m) {
    MaterialPtr km;
    km.ptr = m;
    if (materialTree.find(km)) return false;
    Spair<String, int> p;
    p.first = m->getName();
    p.second = 0;
    Spair<Time, IntPair> tt;
    tt.first = m->getTime();
    // printf("M! %d\n", m->getTime().calHours());
    int cnt = 0;
    if (nameTree.find(p)) {
        // printf("M! 1\n");
        int index = nameTree.search(p).second;
        Material* latest = materials[index];
        while (latest->getNextVersion() != NULL) {
            latest = latest->getNextVersion();
            cnt++;
        }
        latest->setNextVersion(m);
        materialTree.insert(km);
        tt.second.first = index;
        tt.second.second = cnt + 1;
        timeTree.insert(tt);
        return true;
    }
    // printf("M! 2\n");
    materials.pushBack(m);
    mForSort.pushBack(m);
    p.second = materials.getSize() - 1;
    materialTree.insert(km);
    nameTree.insert(p);
    tt.second.first = p.second;
    tt.second.second = 0;
    timeTree.insert(tt);
    // printf("M! %d %d\n", timeTree.find(tt), tt.first.calHours());
    // printf("M! 3\n");
    return true;
}
```

```
}
```

下面分析查找的实现：

根据我们红黑树的查找函数 search(rbNode d)，如果在红黑树中找到了和 d 相等的节点，我们返回这个节点；如果红黑树中不存在和 d 相等的节点，我们返回 d。

由此，我们需要先构建一个查找用的节点 p，其类型为 Spair。first 为待查找的关键字，如果是按名称查找，那么 first 即为需要查询的名称，如果是按时间查找，那么 first 即为需要查找的更新时间。second 为空。

首先先使用红黑树的 find 函数，判断 p 是否存在红黑树中，若不存在，直接返回一个空指针。若存在，则进一步使用 search 函数，以 p 为搜索目标，在红黑树中进行查找。由于我们定义的 Spair 类的比较逻辑，我们会在红黑树中找到一个 first 和 p.first 相同的节点，并返回该节点的值，也就是我们期待找到的节点。这个节点的 second 字段，保存了我们预先存储好的下标和版本信息。最后去 materials 数组中，把对应的文件资料指针返回即可。

代码实现为：

```
Material* Materialsys::getMaterialByName(const string& name) {
    Material* homework = NULL;
    Spair<String, int> p;
    p.first = name;
    p.second = 0;
    if (!nameTree.find(p)) return NULL;
    int index = nameTree.search(p).second;
    homework = materials[index]->getLatestVersion();
    return homework;
}

Material* Materialsys::getMaterialByTime(const Time& t) {
    Material* m = NULL;
    Spair<Time, IntPair> tt;
    tt.first = t;
    // printf("!M %d %d\n", tt.first.calHours(), timeTree.find(tt));
    if (!timeTree.find(tt)) return NULL;
    tt = timeTree.search(tt);
    int index = tt.second.first;
    int index2 = tt.second.second;
    // printf("!M ?%d %d\n", index, index2);
    m = materials[index];
    while (index2--) {
        m = m->getNextVersion();
    }
    // printf("!M %p\n", m);
    return m;
}
```

(3) 按资料名子串（关键字）查询

如果不是按照完整的名称进行搜索，就无法使用红黑树进行。因为红黑树的高效搜索效率是基于它的二叉排序树的性质，但是根据出现子串进行搜索时，包含子串与否和资料的名称不存在严格的顺序关系，不可以进行排序，因此不能利用红黑树作为搜索树的性质来降低时间复杂度，反而会因为递归遍历整棵树，导致效率大幅下降。

我们直接遍历了 materials 数组，通过依次判断每个数组元素的名称是否包含查询的子串来实现按子串查找。询问是否包含则利用了模式串匹配的 KMP 算法，此前已经介绍过。

代码实现如下：

```
vector<Material*> MaterialsSys::search(const string& name) {
    vector<Material*> result;
    for (int i = 0; i < materials.getSize(); i++) {
        KMP kmp(materials[i]->getName(), name);
        if (kmp.isExist()) {
            Material* tmp = materials[i]->getLatestVersion();
            result.pushBack(tmp);
        }
    }
    return result;
}
```

(4) 获得特定版本的资料

传入一个下标表示该资料项在 materials 数组中的位置，再传入一个版本号，表示是第几个版本。直接在 materials 数组下找到链表头，再循环得到想要的版本即可，代码实现如下：

```
Material* MaterialsSys::getCertainVersion(const string& name, int t) {
    Material* m = NULL;
    Spair<String, int> p;
    p.first = name;
    p.second = 0;
    if (!nameTree.find(p)) return NULL;
    int index = nameTree.search(p).second;
    m = materials[index];
    while (t--) {
        if (m) m = m->getNextVersion();
    }
    return m;
}
```

(5) 按照名称或更新时间排序

MaterialSys 支持按照文件资料名称或更新时间对所有课程进行排序，排序方法为快速排序。按更新时间排序接口函数为：

```
void MaterialsSys::sortByTime(int l, int r)
```

按文件资料排序接口函数为：

```
void MaterialsSys::sortByName(int l, int r)
```

返回所有课程的接口函数为：

```
vector<Material*> MaterialsSys::getAllMaterial()
```

这里采用的都是和基础算法描述部分相同的快速排序，代码实现上没有明显区别，具体代码实现可以参看源程序。

7.4.8 考试类 Test

考试类存储一场考试对应的时间和对应的课程，仅仅是提供一个方便操作的数据对象。类的声明如下：

```
class Test {
private:
    Pair<Time, Time> testTime;
    int cid;

public:
    Test();
    Test(Time sTime, Time eTime, int id);
    Pair<Time, Time> getTime(); // 以时间对的形式获得考试的起止时间
    Time getBeginTime(); // 获得开始时间
    Time getEndTime(); // 获得结束时间
    int getcid(); // 获得课程名称
    Test& operator = (const Test &other);
};
```

基本的接口函数都比较简单且直观，这里就不单独说明了。代码实现可以参看源程序。

7.4.9 课程类 Course

课程类 Course 用于存储课程的基本信息。包括课程ID，课程名称，课程地点校区，课程教室，课程起止时间，课程起止周数，提交和待提交的作业等。类的声明如下：

```
class Course {
private:
    int courseid; // 课程编号，是用来方便查找和调用的
    String courseName; // 课程名
    int location; // 建筑ID，课程地点的int描述
    int campus; // 校区ID，校区的int描述
    String teacher;
    String qq;
    String classroom; // 课程地点的字符串描述
    Pair<Time, Time> courseTime; // 课程时间，包括开始和结束
    Pair<int, int> courseweek; // 课程周数，包括开始和结束
    Test test; // 这里放考试信息
    Vector<String> handIn; // 已经提交的作业
    Vector<String> toBeHandIn; // 待提交的作业
}
```

类的外部接口函数包括：

```
class Course {
public:
    Vector<itinerary*> events;
    Course();
    ~Course();
    Course(int id, String name, int loc, int cps, String cls,
           Pair<Time, Time> ctime, Pair<int, int> cweek);
    // 构造函数，允许传入一个课程的基本信息，生成一个对应的课程
```

```

explicit Course(const Course &other);
void addTest(Test t); // 设置课程的考试信息，传入一个考试项
void setTest(Time sTime, Time eTime); // 设置课程的考试时间，会自动生成一个考试项
Test getTest(); // 获得课程的考试信息
int getId(); // 获得课程的 ID
String getName(); // 获得课程的名称
int getIntLoc(); // 获得课程地点的 ID
void setIntLoc(const int v); // 设置课程地点的 ID
String getStringLoc(); // 获得课程地点，返回课程地点的字符串描述
void setStringLoc(const String v); // 设置课程地点，传入一个课程地点的字符串描述
Pair<Time, Time> getTime(); // 获得课程开始结束时间
void setTime(int bg, int ed); // 设置课程开始结束时间
Time getBeginTime(); // 获得开始时间
Time getEndTime(); // 获得结束时间
int getBeginWeek(); // 获得起始周
int getEndWeek(); // 获得结束周
Pair<int, int> getweek(); // 获得课程的起止周数，得到的是一个周数对
void setweek(int bg, int ed); // 设置课程的起止周数
void addHomework(String name); // 添加作业，传入一个作业名称，自动生成一个对应的未提交作业
void handInHomework(String name); // 提交作业，传入对应作业的名称，标记其为已提交
Vector<String> getToBeHandInHomework(); // 获得所有已提交的作业名称列表
Vector<String> getHandInHomework(); // 获得所有未提交的作业名称列表
Course& operator = (const Course &other);
};


```

主要介绍一下提交作业函数 handInHomework，其余函数比较简单，直接阅读源代码即可。提交作业函数代码实现如下：

```

void Course::handInHomework(String name) {
    for (int i = 0; i < toBeHandIn.getSize(); i++) {
        if (name == toBeHandIn[i]) {
            Basic :: swapElement(&toBeHandIn[i],
                &toBeHandIn[toBeHandIn.getSize() - 1]);
            toBeHandIn.popBack();
            handIn.pushBack(name);
            break;
        }
    }
}

```

我们 for 循环访问 toBeHandIn 数组，即未提交的作业列表。如果找到了当前提交的作业，因为 Vector 类只支持从尾部删除，不支持任意位置删除，所以我们用未提交作业列表的最后一项和当前项交换，再 popBack 删除最后一项即可。在实际应用场景下，一门课程待提交的作业一般不会很多，所以我们可以直接遍历整个未提交作业数组来找到具体提交的是哪一项作业；在数据范围较小的情况下，如果引入平衡树来做此类查找，反而因为平衡树的时间复杂度存在较大的常系数，导致查找效率降低。

7.4.10 课程系统 CourseSys

课程系统 CourseSys 是课程管理系统模块后端最外部的封装之一，提供了管理课程系统的接口，直接和前端进行对接。其类定义如下：

```
class CourseSys {
private:
    AVL<Course_ptr> courseTree;
    int cnt;
    Vector<Course*> courseArray;

public:
    CourseSys();
    ~CourseSys();

    void addCourse(String name, String cls, int loc, int cps,
                   Time btime, Time etime, int bweek, int eweek);
    // 添加课程，传入课程的基本信息后，会自动添加课程，并插入 AVL 树中
    Vector<String> getAllCourseNames(); // 获得全部课程的名称
    Course* getCourseByName(const String& name); // 通过名称搜索课程
    Course* getCourseById(int id); // 通过 id 查找课程
    Vector<Course*> getAllCoursePtr(); // 获得所有课程，返回的是指针数组
    void sortId(int l, int r); // 按照课程的id进行排序
    void sortName(int l, int r); // 按照课程的名称进行排序
};
```

在课程系统 CourseSys 类中，定义了三个成员对象：用来维护所有课程的 AVL 平衡树 courseTree，用来存储所有课程指针并提供便捷访问的指针数组 courseArray，记录课程数量的整型 cnt。其中 AVL 树的数据类型未 Course_Ptr，这是一个课程类 Course 的指针类，重载了基于课程名称比大小的比较运算符，便于 AVL 树进行比较和排序。

对于课程系统，和文件系统类似，我们也采用了平衡树对其进行维护。因为相较于文件而言，实际应用场景中，一学期需要学习的课程总数并不多，因此，使用 AVL 树也能很好地达成我们去重和查找的要求。

成员函数为前端的调用提供了便捷的接口，其中比较重要的接口函数有：

(1) 添加课程 addCourse

这个函数允许用户传入课程的必要基本信息，之后会自动动态声明一个新的课程，并在判重后加入 AVL 树和指针 Vector 中。

代码实现如下：

```
void CourseSys :: addCourse(String name, String cls, int loc, int cps,
                             Time btime, Time etime, int bweek, int eweek) {
    Pair<Time, Time> ctime;
    Pair<int, int> cweek;
    ctime.first = btime;
    ctime.second = etime;
    cweek.first = bweek;
    cweek.second = eweek;
    Course* newCourse = new Course(cnt++, name, loc, cps, cls, ctime, cweek);
    Course_ptr nc;
    nc.ptr = newCourse;
    if (courseTree.exist(*courseTree.getRoot(), nc)) {
        cnt --;
```

```

    delete newCourse;
    return;
}
courseArray.pushBack(newCourse);
courseTree.insert(courseTree.getRoot(), nc);
}

```

首先程序会根据传入的课程基本信息，new 一个课程类，并为其生成搜索用的关键字 nc。接着用刚刚生成的 nc，在 AVL 树中搜索是否存在课程名相同的元素，若存在，说明当前课程已经加入过了，直接退出。若不存在，则说明当前课程没有发生重复，将其加入 AVL 树和指针 Vector 中。

(2) 按名称搜索课程 getCourseByName

这个函数支持传入具体的课程名称，返回一个对应课程的指针，如果不存在这样的课程，返回空指针。代码实现如下：

```

Course* CourseSys::getCourseByName(const String& name) {
    // 在AVL中搜索
    Pair<Time, Time> ctime;
    Pair<int, int> cweek;
    String cls;
    Course* srh = new Course(-1, name, 0, 0, cls, ctime, cweek);
    Course_ptr nc;
    nc.ptr = srh;
    Course_ptr ans;
    courseTree.exist(*courseTree.getRoot(), nc);
    ans = courseTree.search(*courseTree.getRoot(), nc);
    if(ans.ptr->getId() == -1) return NULL;
    return ans.ptr;
}

```

这里首先生成了搜索用关键字 nc，然后在 AVL 树种进行搜索获得需要找的课程。因为 AVL 树本身就是按照名称比较的二叉搜索树，所以利用这一点在 AVL 树中进行搜索效率更高。

(3) 排序函数

这里采用了快速排序对所有课程进行排序，具体实现和前面快速排序类似，具体参见源代码部分。

7.5 模块测试数据及结果

添加课程

输入：

UI

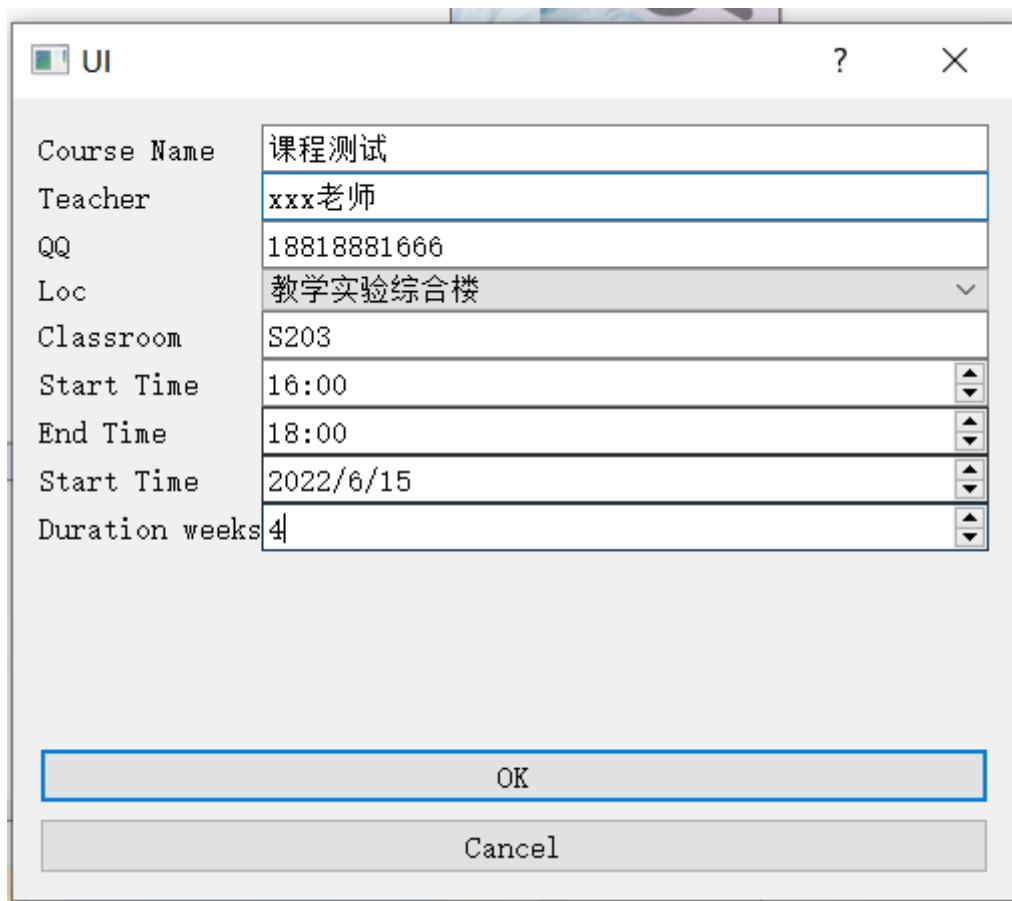
?

X

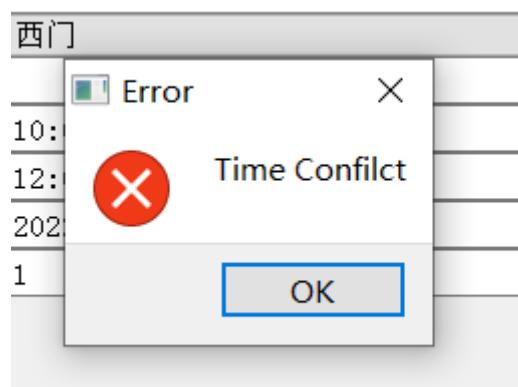
Course Name	课程测试
Teacher	xxx老师
QQ	18818881666
Loc	教学实验综合楼
Classroom	S203
Start Time	16:00
End Time	18:00
Start Date	2022/6/15
Duration weeks	4

OK

Cancel



输出 (时间冲突) :



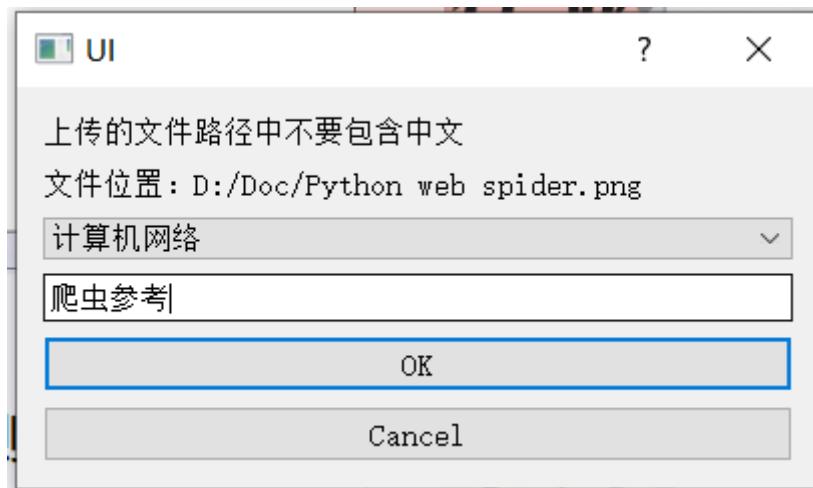
输出 (添加成功并可视化) :

请输入查询活动的名称

	Mon.	Tue.	Wed.	Thu.
15:00	活动闹钟测试 沙河校区 综合办公楼 203			
16:00	个人活动		课程测试 沙河校区 教学实验综合楼 S203 课程或考试	
17:00				
18:00				

上传文件

输入：

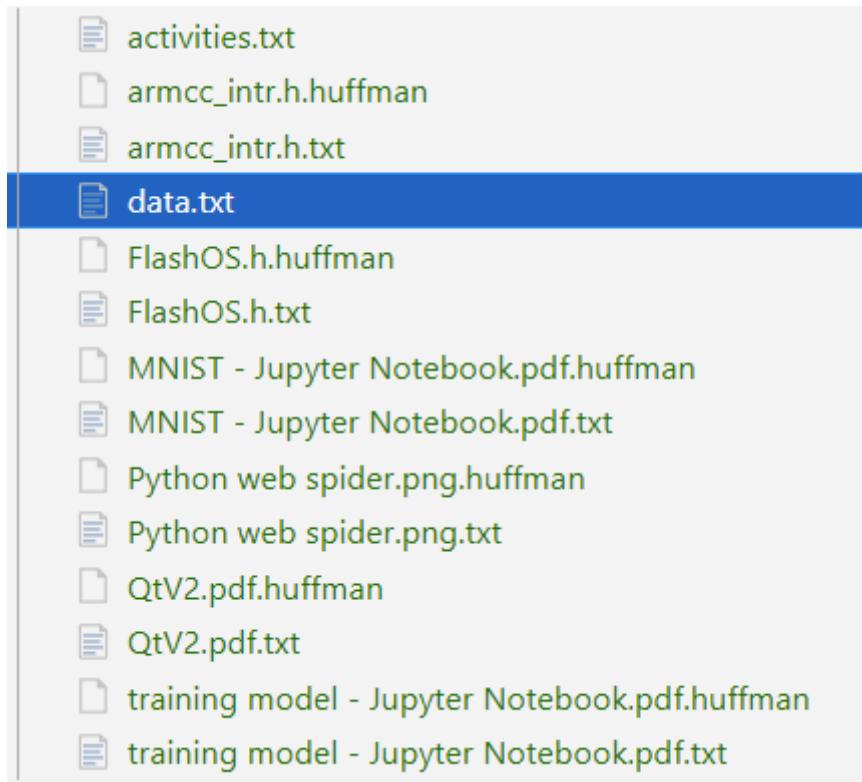


输出：

```

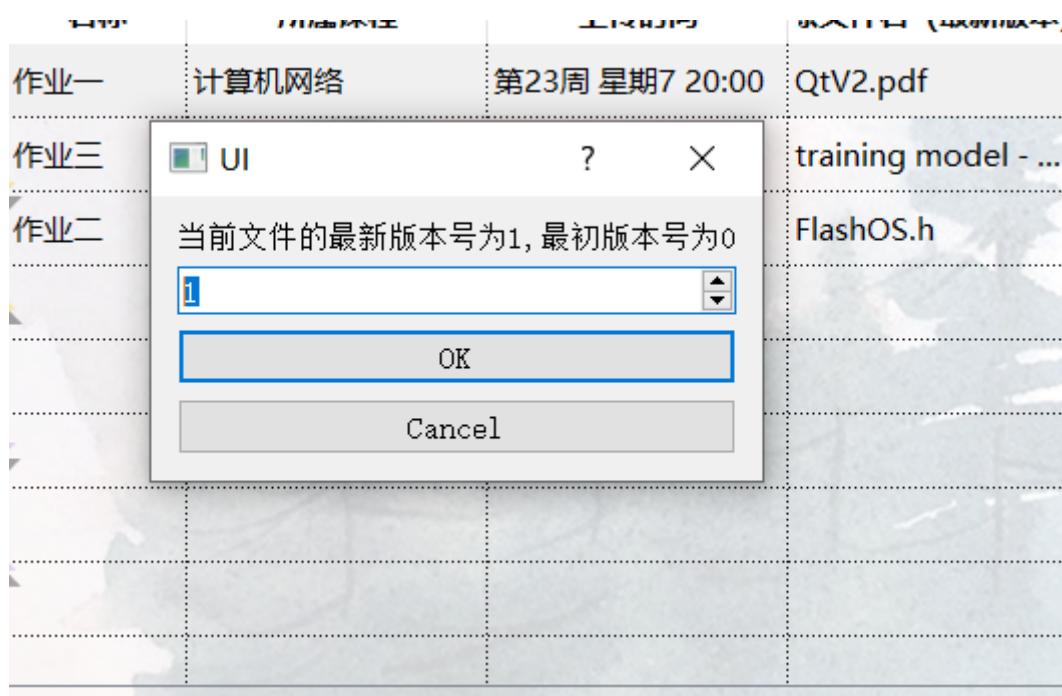
26 计算机网络·0
27 爬虫参考
28 D:/project/CourseAuxiliarySystem/user/testuser/Python web spider.png
29 150·70·87·130·171·242·151·245·84·198·33·21·153·151·51·168
30 4·4·24
31

```



下载文件

输入：



输出：

> Data (D:) > project

名称	修改日期	类型	大小
build-CourseAuxiliarySystem-Desktop...	2022/6/14 15:07	文件夹	
build-logisticsManagement-Desktop...	2022/6/10 12:25	文件夹	
build-logisticsManagement-Desktop...	2022/5/19 21:24	文件夹	
build-logisticsManagement-Desktop...	2022/5/19 21:24	文件夹	
course	2022/3/24 22:29	文件夹	
CourseAuxiliarySystem	2022/6/11 19:05	文件夹	
DNSrelay	2022/5/5 9:16	文件夹	
GraphBuilder	2022/4/7 20:42	文件夹	
KichenManager	2022/2/23 20:31	文件夹	
logisticsManagement	2022/5/2 22:00	文件夹	
main	2022/5/30 21:13	文件夹	
res	2022/4/10 15:36	文件夹	
UI	2022/4/13 12:39	文件夹	
CourseAuxiliarySystem.zip	2022/6/14 9:45	ZIP 压缩文件	64,192 KB
QtV2.pdf	2022/6/14 20:57	WPS PDF 文档	2,752 KB

文件排序

上传时间排序：

名称排序

按时间排序

● 按名称排序

8 时间模拟系统

时间模拟系统主要在前端实现，使用了qt的QDateTime以及 QTimer 库完成。

8.1 时间初始化

对于模拟实践的初始化，我们让程序在运行初始便通过库函数 `[static] QDateTime QDateTime::currentDateTime()` 完成

QDateTime QDateTime:: [static] currentDateTime()

Returns the current datetime, as reported by the system clock, in the local time zone.

8.2 时间推进

在要求中提到默认情况下按照10s为1小时的速度进行时间推进，进而为了更精确的显示时间并让使用者能较为及时的感受到时间的变化，系统中转换成每1s模拟时间推进6s。而这是通过 QTimer (该库为 qt 提供的计时器) 实现的，同样是在系统开始运行时便设置计时器的触发间隔为1000ms(1s)，每次到时则会触发信号 `[signal] void QTimer::timeout()`，并随后通过槽函数进行时间的修改来实现时间推进功能，并同样是通过槽函数完成界面显示上的时间更改。

```
timer = new QTimer();
timer->setInterval(1000);
connect(timer, &QTimer::timeout, this, [=]() {
    curTime = curTime.addSecs(360);
    setTime(false);
    emit sendTime();
});
timer->start();
```

8.3 界面间时间统一

除了主界面，系统的每个界面均设有两个私有成员，分别是：`QDateTime *time;` 以及 `QTimer *timer;`，会在界面初始化时将这两个成员传入，因此各个界面的时间和计时器其实均为同一个变量，是主界面时间以及计时器的副本，并通过相关信号以及槽函数的连接来对子界面的时间进行修改。

```
//界面初始化时传入计时器以及时间的指针
//通过信号和槽在每次触发计时器时更改子界面的时间显示
clock = new clockwidget(&curTime, timer);
connect(this, &Widget::sendTime, clock, [=]() {
    clock->setTime(false);
});
connect(this, &Widget::sendTime, clock, &clockwidget::checkAlarm);
guide = new guidewidget(&curTime, timer);
connect(this, &Widget::sendTime, guide, [=]() {
    guide->setTime(false);
});
```

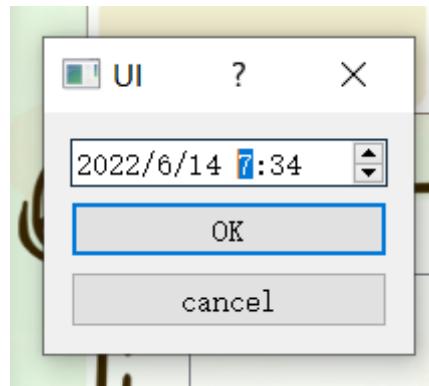
```

calander = new calanderwidget(&curTime, timer);
connect(this, &widget::sendTime, calander, &calanderwidget::setTime);
connect(calander, &calanderwidget::setAlarm,
        clock, &clockwidget::setActivityClock);
manage = new manageWidget(&curTime, timer);
connect(this, &widget::sendTime, manage, [=]() {
    manage->setTime(false);
});

```

8.4 时间加速

在主界面有时间加速按钮，在弹出的输入框中输入期望的时间便可以直接对模拟时间进行更改。



8.5 人机交互时间暂停

只需要在交互开始时让计时器暂停，在交互结束后让计时器重启即可。

```

void widget::changeTimePage() {
    timer->stop(); //暂停计时器
    QDialog *timeEdit = new QDialog();
    timeEdit->setAttribute(Qt::WA_DeleteOnClose);
    QVBoxLayout *mainLayout = new QVBoxLayout(timeEdit);
    QPushButton *okbtn = new QPushButton("OK", timeEdit);
    QPushButton *cancelbtn = new QPushButton("cancel", timeEdit);
    QDateTimeEdit *edit = new QDateTimeEdit(curTime, timeEdit);
    mainLayout->addWidget(edit);
    mainLayout->addWidget(okbtn);
    mainLayout->addWidget(cancelbtn);
    timeEdit->show();
    connect(okbtn, &QPushButton::clicked, this, [=]() {
        curTime = edit->dateTime();
        setTime(true);
        timeEdit->close();
        timer->start(1000); //重启动计器
    });
    connect(cancelbtn, &QPushButton::clicked, timeEdit, [=]() {
        timeEdit->close();
        timer->start(1000); //重启动计器
    });
}

```

时间模拟系统运行情况可以在前端直观读出，故测试略。

9 日志文件系统

日志系统同样在前端实现，我们设置了一个简单的全局函数以及全局数组便于日志的生成以及存储。

```
void addLog(QDateTime curTime, QString message) {
    QString log;
    log += QDateTime::currentDateTime().toString("MM.dd HH:mm") + ' ';
    log += curTime.toString("MM.dd HH:mm") + ' ';
    log += message;
    record.push_back(log);
}
```

从该全局函数中可以看出日志的格式为：

实际时间 模拟时间 日志内容

产生的(部分)日志如下图所示：

```
06.12 17:26 06.12 17:50 使用了导航 1 0->0 17
06.12 17:26 06.12 17:50 使用了导航 1 0->0 20
06.12 17:26 06.12 17:50 使用了导航 1 0->0 3
06.13 10:54 06.13 11:41 添加活动：活动闹钟测试
06.13 10:54 06.13 11:41 添加了闹钟 15:00
06.13 22:36 06.13 23:05 使用了导航 0 1->1 27
06.13 22:36 06.13 23:05 使用了导航 0 1->1 27
06.13 22:36 06.13 23:05 使用了导航 0 1->1 26
06.13 22:36 06.13 23:05 使用了导航 0 1->1 28
```

同时，在本次程序运行时产生的日志通过点击主界面日志文件按钮可以进行查看，效果如图：



UI

?

X

06.14 00:29 06.14 04:28 课程:面向对象程序设计 考试定于06.30日10:00-



10 整体测试运行

10.1 开始运行和登录

双击“exe”文件，开始运行软件。首先会打开开始界面，开始界面如下：



鼠标单击后，会进入登录界面。登录界面如下：



在左侧输入框中输入账号和密码，单击确认键，进入学生主界面。



10.2 时间加速功能测试

单击左侧“时间加速”按钮，会弹出时间修改的对话框。在这里修改时间：



修改时间结果为：



10.3 闹钟功能测试

点击左上的闹钟图标，可以进入闹钟界面。

闹钟管理

0 4 : 3 8

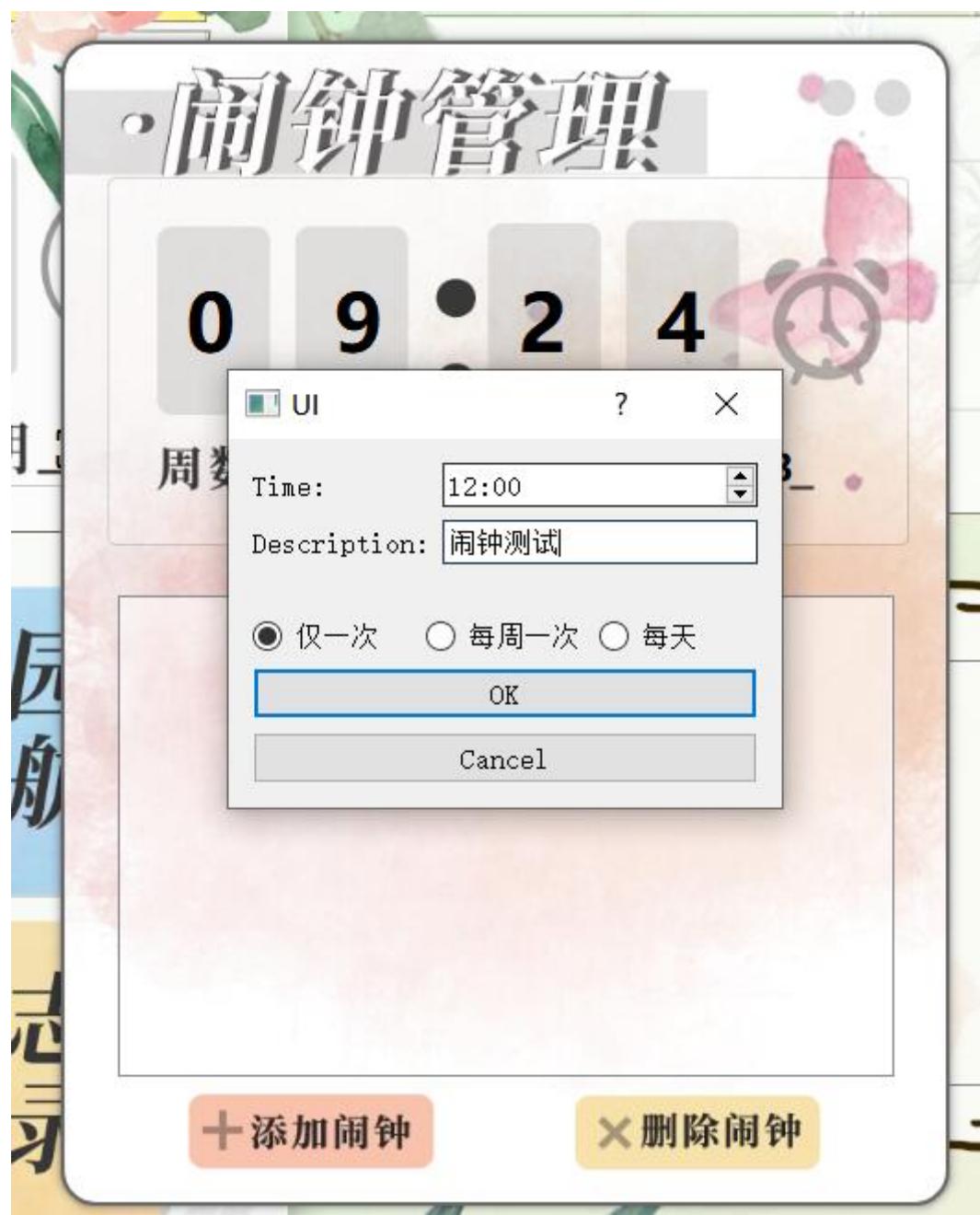


周数: 24 6月 16日 星期 4

+ 添加闹钟

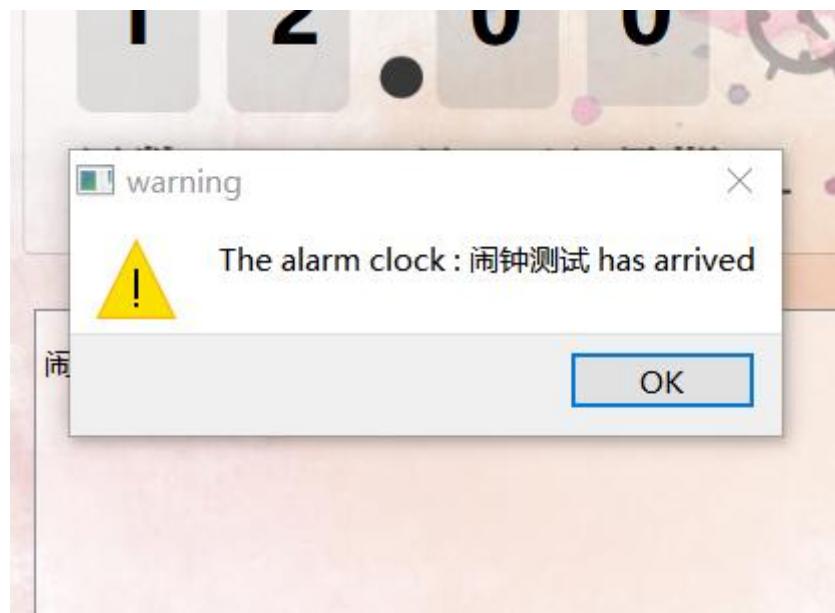
× 删除闹钟

点击左下角添加闹钟按钮，会弹出添加闹钟的对话框：



点击 OK 按钮，可以看到成功添加了闹钟。

当时间到达闹钟时间时，系统会发出提示：





点击左下角课程管理按钮，可以进入课程管理界面。

10.4 课程管理功能测试

进入课程管理子界面，进行功能测试。

课程管理界面的左上方显示了用户当前选中的课程的详细信息，下方则是课程的详细列表，用户选中对应课程，就会在上方展示详细信息。右侧有四个按钮，选中作业按钮，会在右侧显示当前所有作业。

点击“资料”按钮，右侧界面会切换到资料页，在这里可以查看到所有的资料。

课程管理

WEEK: _24_ DAY: _15_

当前课程

面向对象程序设计
上课时间: 13:00
下课时间: 15:00
上课地点: 教学实验综合楼 N210
QQ群为: 00005753200
老师为: 9老师

计算机网络课程设计 数字逻辑 面向对象课程设计 程序设计

添加课程 提交作业 上传资料

作业 资料 考试 通知

请输入文件名/资料名

名称	所属课程	上传时间	原文件名 (最新版本)
第一章资料	计算机网络	第23周 星期7 5:00	armcc_intr.h

○ 按时间排序 ○ 按名称排序

点击“考试”按钮，右侧界面切换为考试信息。在这里可以读出考试的详细信息。

课程管理

WEEK: _24_ DAY: _15_

当前课程

当前无课程

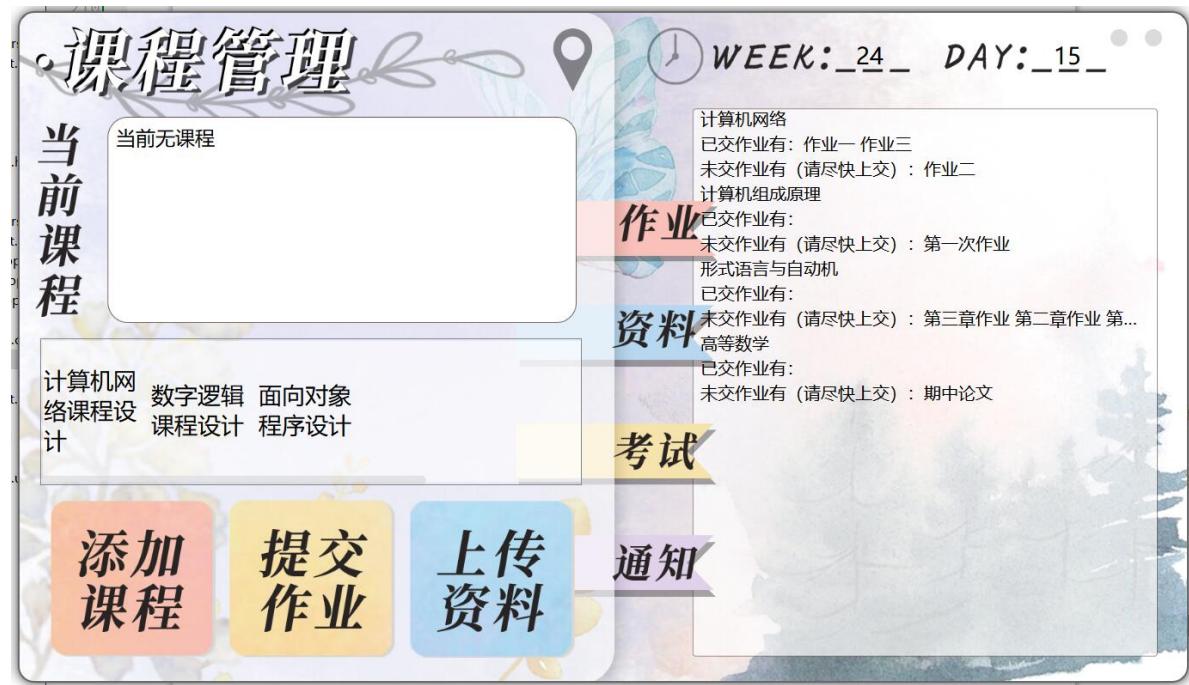
计算机网络课程设计 数字逻辑 面向对象课程设计 程序设计

添加课程 提交作业 上传资料

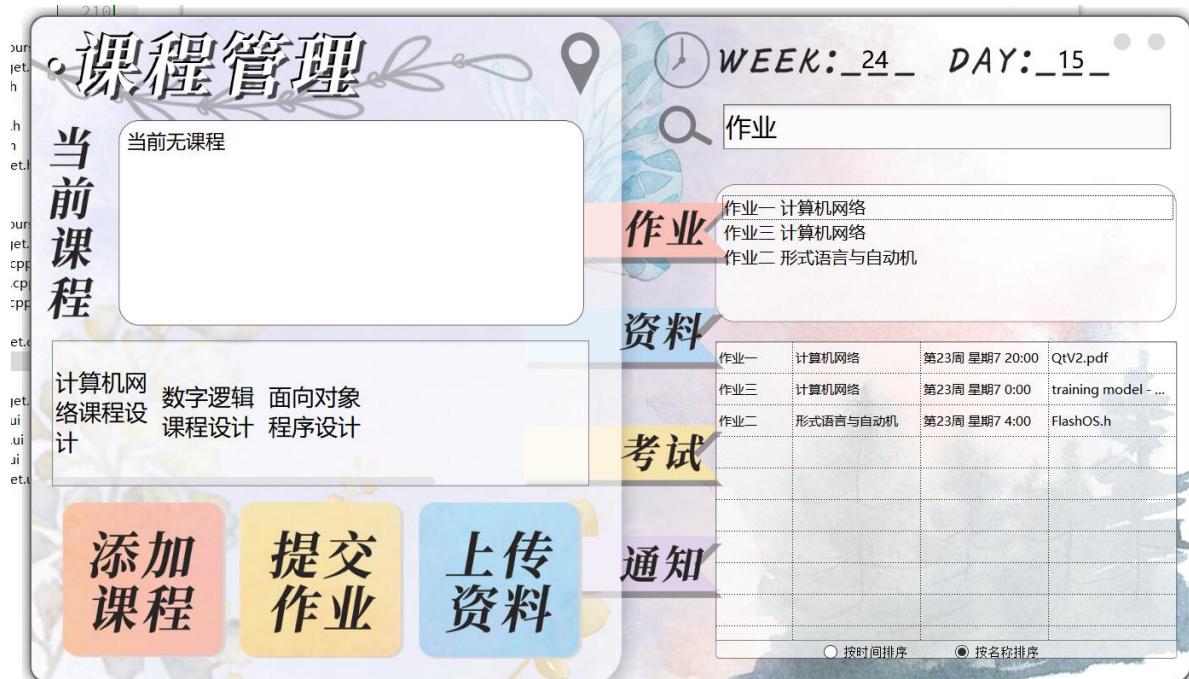
作业 资料 考试 通知

计算机网络考试时间为:
第28周 星期1 10:00-12:00
计算机组成原理考试时间为:
第28周 星期2 8:00-10:00

点击“通知”按钮，右侧切换为通知界面，在这里会显示所有未交作业的提醒。



点击“作业”按钮，回到作业界面，在这里可以从左上角的搜索框输入作业名称，会在下方列出所有含有该关键字的作业。



点击“添加课程”按钮，可以进行课程的添加。会弹出对应的对话框，设置相关信息后完成课程的添加：

UI

?

X

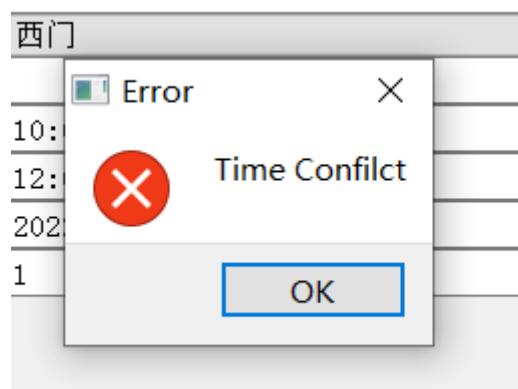
Course Name	课程测试
Teacher	xxx老师
QQ	18818881666
Loc	教学实验综合楼
Classroom	S203
Start Time	16:00
End Time	18:00
Start Time	2022/6/15
Duration weeks	4

OK

Cancel

This screenshot shows a Windows-style dialog box titled "UI". Inside, there's a table with course information: Course Name (课程测试), Teacher (xxx老师), QQ (18818881666), Loc (教学实验综合楼), Classroom (S203), Start Time (16:00), End Time (18:00), Start Time (2022/6/15), and Duration weeks (4). At the bottom are "OK" and "Cancel" buttons.

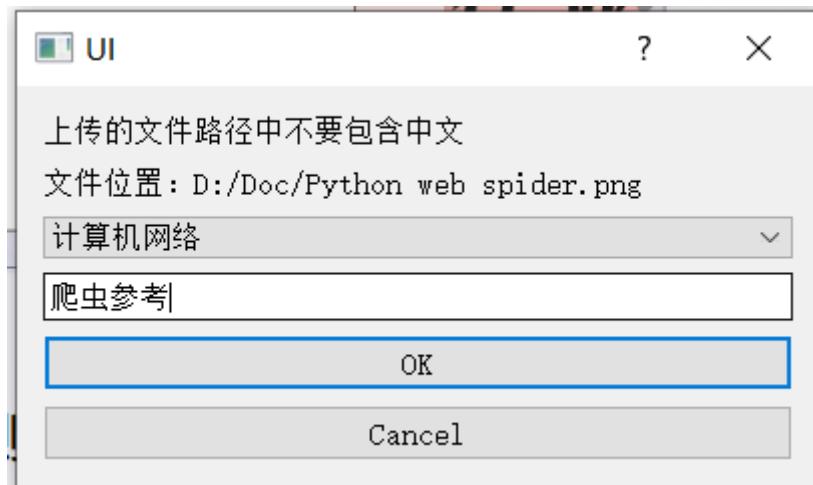
如果正添加的课程与已有课程冲突，会弹出警告：



输出 (添加成功并可视化) :

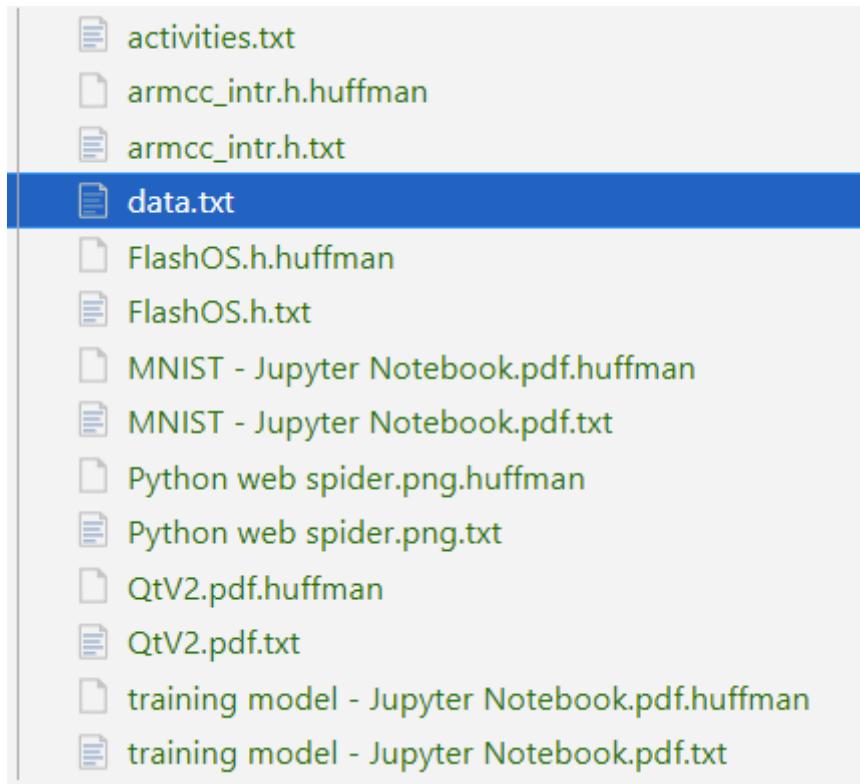


点击“上传资料”按钮可以进行资料的上传，弹出对话框如图：

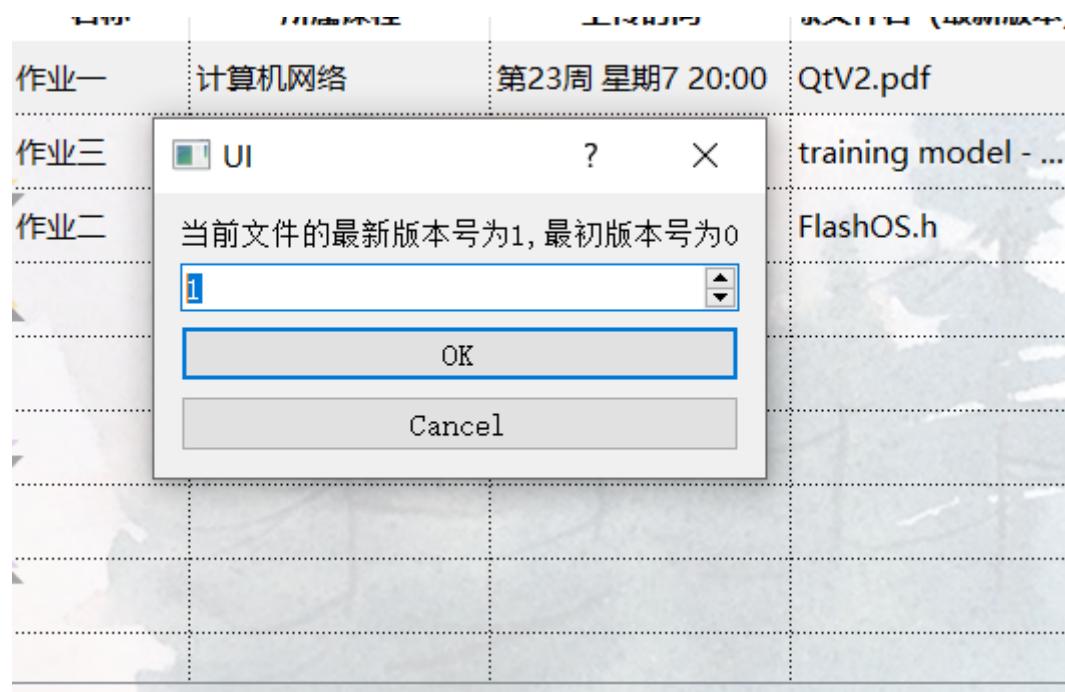


可以看到系统反馈给出计算得到的 MD5 上传时间等信息：

```
26 计算机网络·0
27 爬虫参考
28 D:/project/CourseAuxiliarySystem/user/testuser/Python web spider.png
29 150·70·87·130·171·242·151·245·84·198·33·21·153·151·51·168
30 4·4·24
31
```



在资料部分右侧选中资料，可以选择下载版本进行下载：



在本地文件中看到已经成功下载的资料：

> Data (D:) > project

名称	修改日期	类型	大小
build-CourseAuxiliarySystem-Desktop...	2022/6/14 15:07	文件夹	
build-logisticsManagement-Desktop...	2022/6/10 12:25	文件夹	
build-logisticsManagement-Desktop...	2022/5/19 21:24	文件夹	
build-logisticsManagement-Desktop...	2022/5/19 21:24	文件夹	
course	2022/3/24 22:29	文件夹	
CourseAuxiliarySystem	2022/6/11 19:05	文件夹	
DNSrelay	2022/5/5 9:16	文件夹	
GraphBuilder	2022/4/7 20:42	文件夹	
KichenManager	2022/2/23 20:31	文件夹	
logisticsManagement	2022/5/2 22:00	文件夹	
main	2022/5/30 21:13	文件夹	
res	2022/4/10 15:36	文件夹	
UI	2022/4/13 12:39	文件夹	
CourseAuxiliarySystem.zip	2022/6/14 9:45	ZIP 压缩文件	64,192 KB
QtV2.pdf	2022/6/14 20:57	WPS PDF 文档	2,752 KB

对于上传到系统的文件，可以采用多种方法进行排序

按上传时间排序：

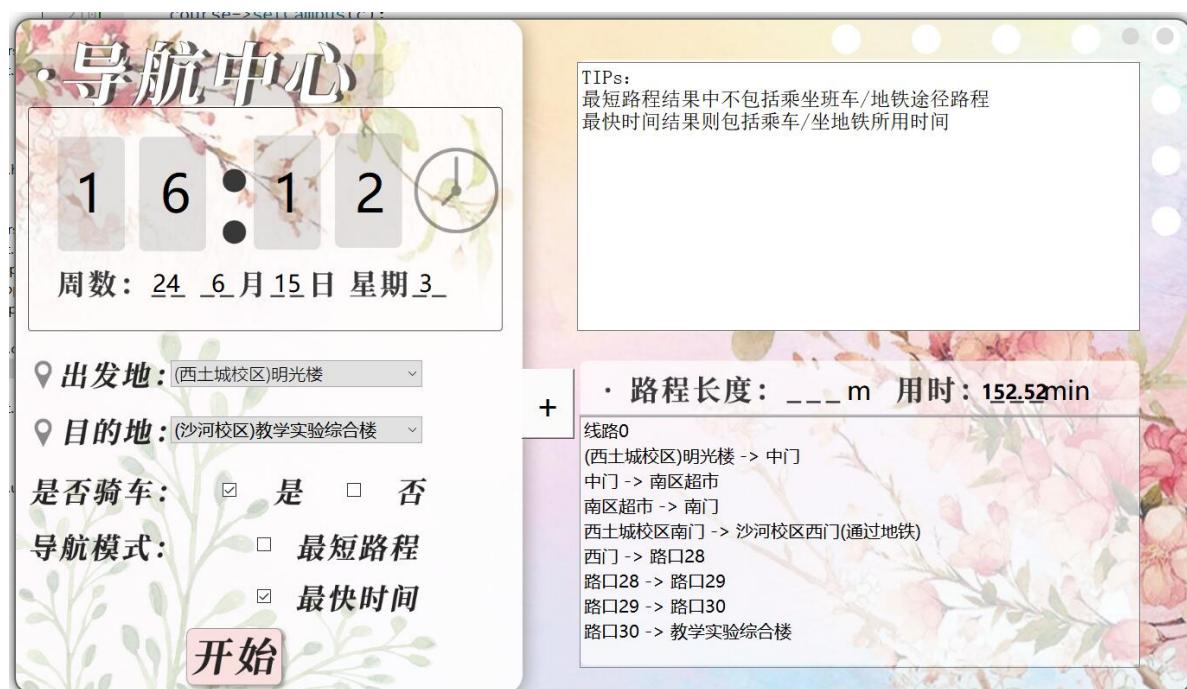
按名称排序：

10.5 校园导航功能测试

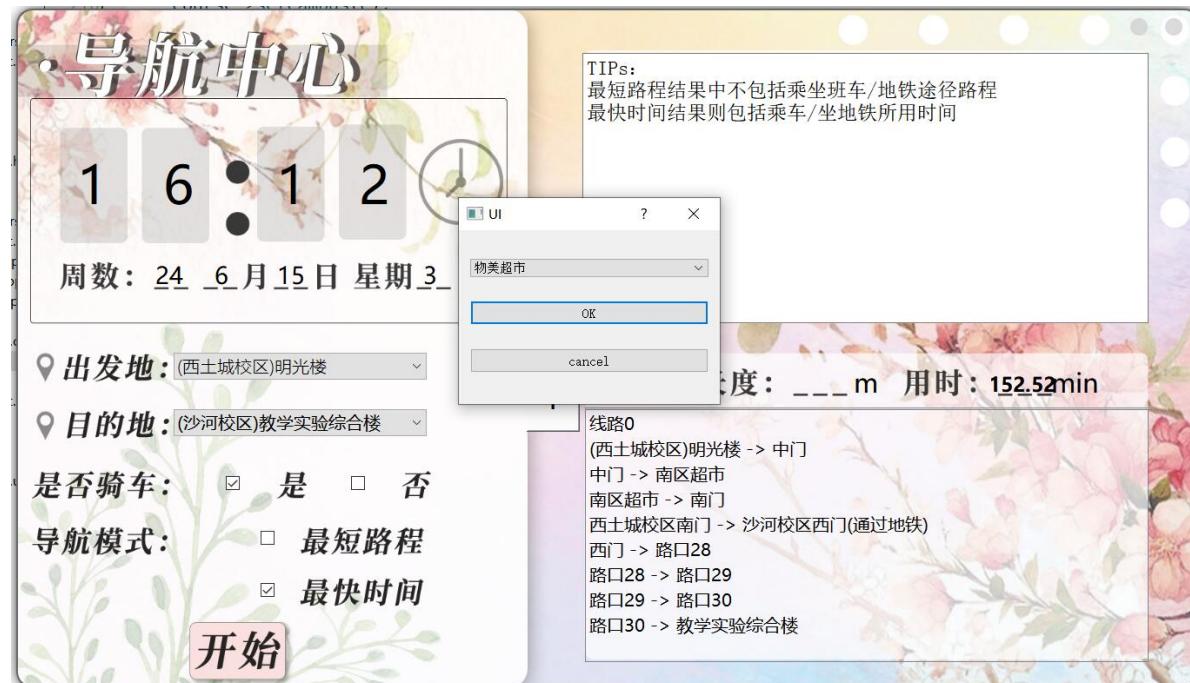
在导航界面，用户可以通过左侧菜单栏设置出发地、目的地以及导航策略

首先是跨校区的导航，输入为骑车、最快时间模式，从西土城-明光楼到沙河-教综楼

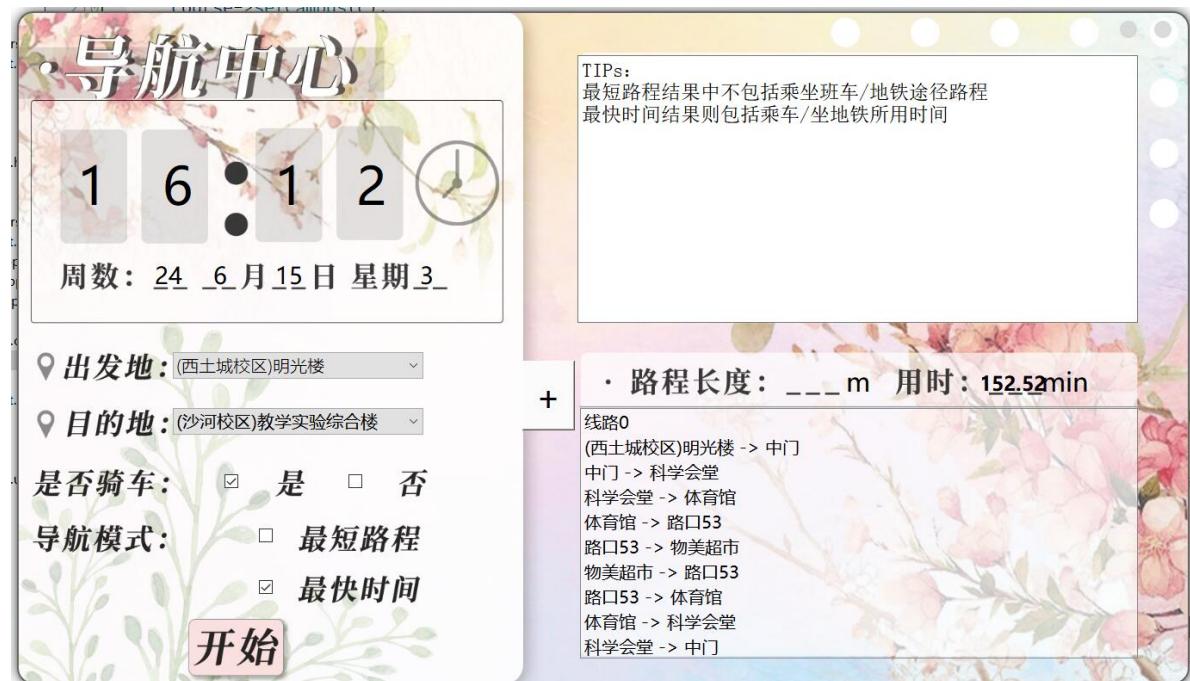
输出一条路径，用时152.52min



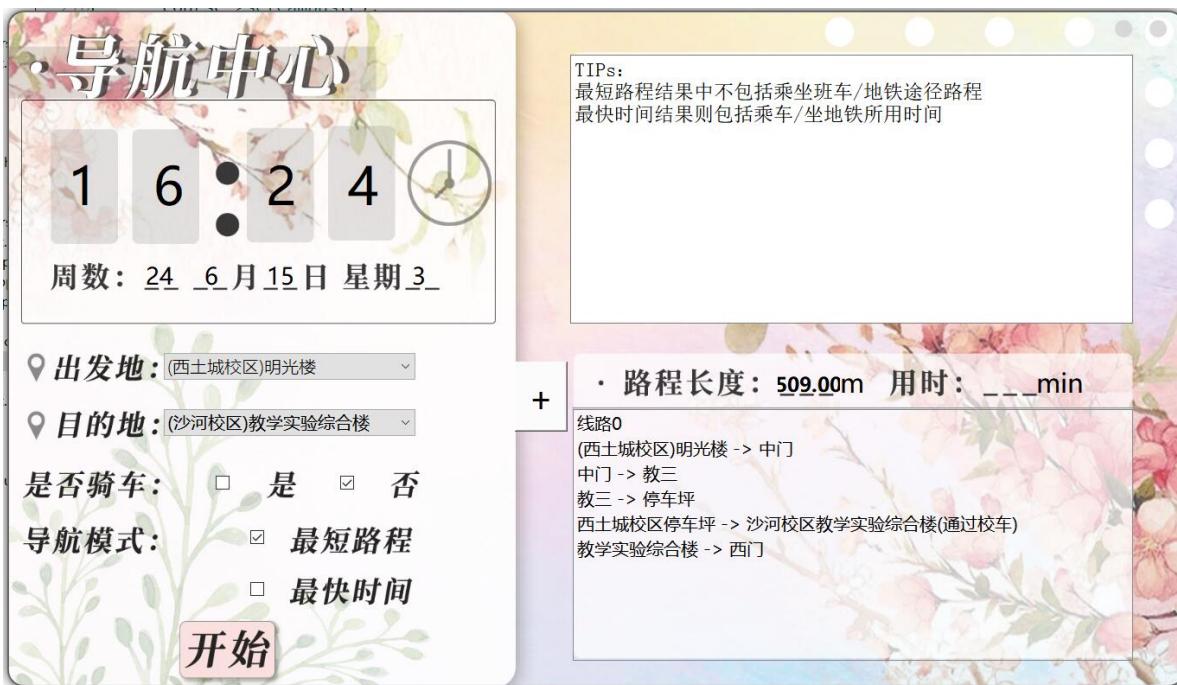
接下来点击左侧边栏边上的“+”，添加途经点，我们添加了物美超市作为途经点。



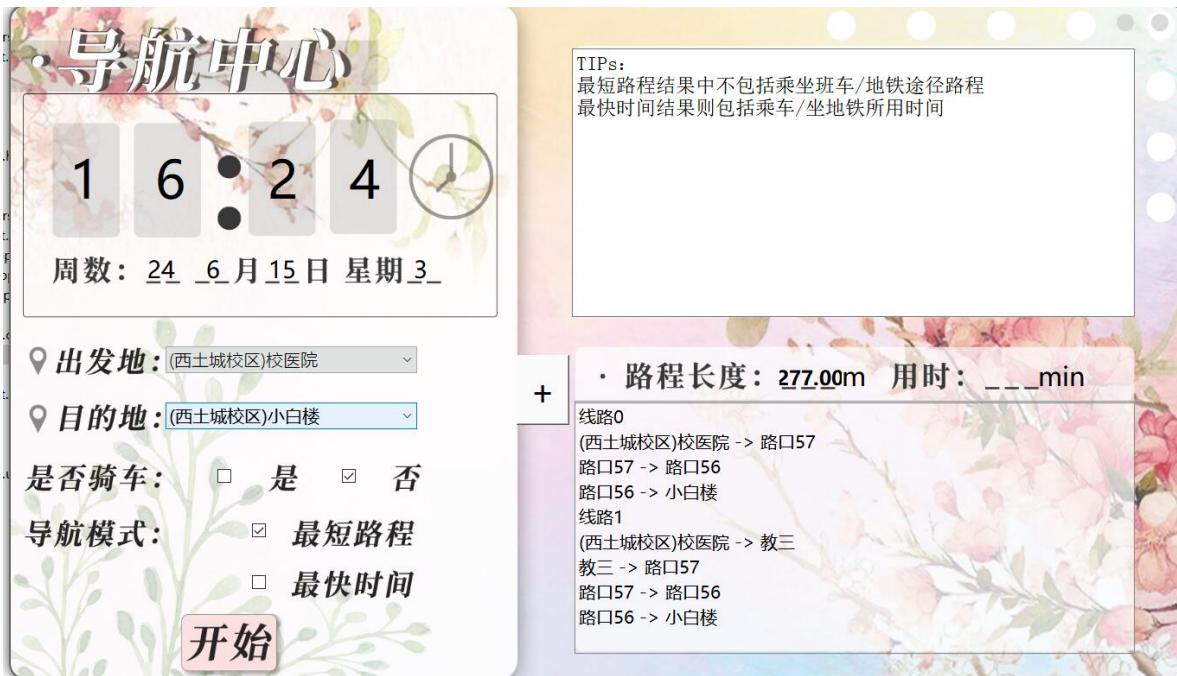
得到新的最短路径。



接下来以最短路程模式导航，得到长为509m的路径（不计算班车路程）。



接下来测试存在多条最短路径的情况:



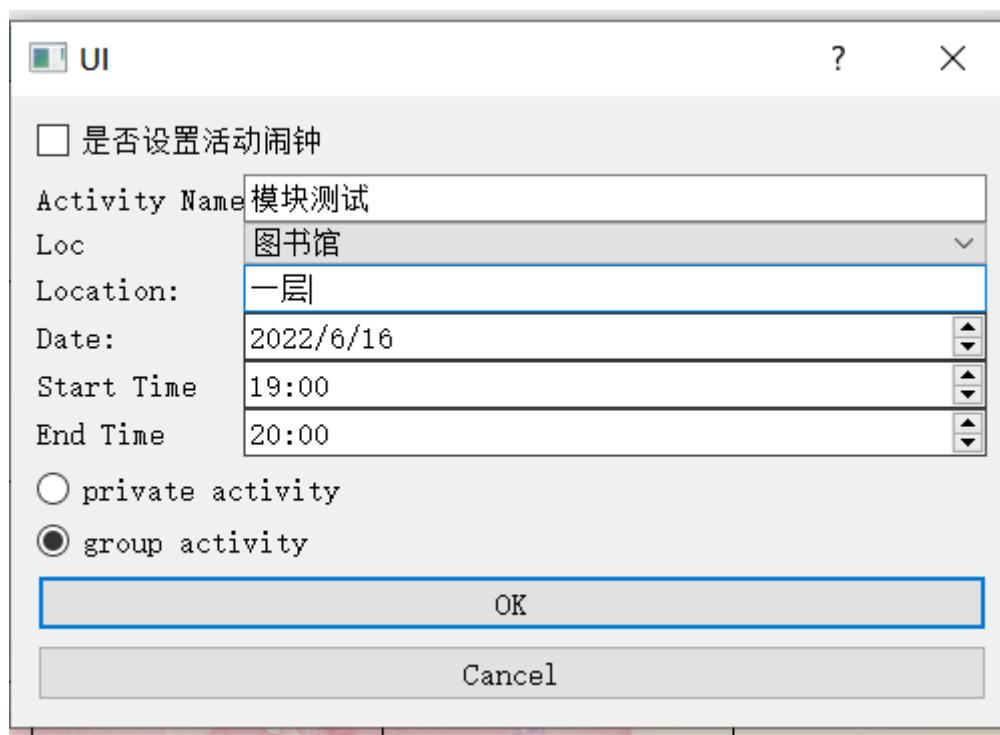
从校医院到小白楼有两条最短路径，都成功输出。

10.6 日程活动功能测试

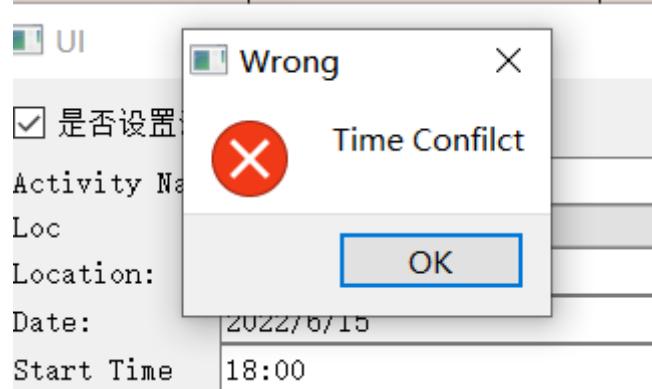
进入日程表界面:



单击右下角的“+”可以实现活动的添加，弹出添加活动的窗口如下：



如果我们设置的活动存在时间冲突，系统会自动检测并提示



如果我们设置的活动正确，那么我们的活动会加入日程表中：

	Mon.	Tue.	Wed.	Thur.
17:00				
18:00				
19:00	班会 沙河校区 学生活动中 心 116 集体活动			
20:00			模块测试日程 沙河校区 运动场 跑道 集体活动	模块测试 沙河校区 图书馆 一层 个人活动

日程表还可以查看非本周的日程，通过上部两个小三角进行周数的切换，当查看非本周日程时，周数显示为红色：

10.7 管理员权限测试

回到学生界面主窗口，单击右下角的“管理员模式”按钮，进入管理员模式。

在管理员模式下，点击“作业情况”按钮，可以发布新作业，弹出窗口如图。我们输入作业名称，点击 OK，作业发布。



我们进入学生权限下的课程管理界面，可以看到计算机网络已经添加了新作业四。

接下来，管理员点击考试发布按钮，在弹出的界面设置待发布的考试的详细信息。



切换到学生模式，学生可以主界面右下角看到考试的通知。



打开课程管理界面-考试，可以看到管理员新发布的考试。



接下来测试通知发布功能，点击通知发布，管理员可以修改课程的地点、时间。



学生打开日程管理，可以看到本周的“线性代数”课程时间已调整。

日程表

周数: 25_6月22日 星期3

活动查询

个人 集体

Mon.	Tue.	Wed.	Thur.	Fri.	Sat.	Sun.
合楼 S510 课程或考试						
12:00						
13:00	毛概 沙河校区 教学实验综 合楼 S308 课程或考试	面向对象程序设计 沙河校区 教学实验综 合楼 N210 课程或考试		线性代数 西土城校区 主楼 N508 课程或考试		
14:00						
15:00						
16:00						+
17:00						

11 评价和改进意见

11.1 自我评价

在本次数据结构课程设计中，我们根据上学期所学的算法与数据结构，和自己课外学习的知识，设计并实现了线下课程辅助系统。此系统包括校园导航、日程管理、课程管理、时间模拟和日志文件五个模块，各模块均充分实现了各自的功能。我们了解掌握了软件开发的方法技巧，并将学过的数据结构与算法知识加以运用：

对于基础的底层算法与数据结构，我们实现了一维数组 Array、二维数组 DyadicArray、变长数组 Vector、字符串 String、二元组 Pair 等基本数据结构。在校园导航模块，我们使用了 Dijkstra 单源最短路算法、堆、动态规划算法、二分查找、深度优先搜索等算法；在日程管理模块，我们使用了线段树、快速排序等算法与数据结构；在课程管理模块，我们使用了 Huffman 树、红黑树、AVL 树、KMP、MD5、链表等算法与数据结构。

11.1.1 课程信息管理和查询

(1.1) 课程数目要求是否满足

满足要求，课程有11门，并有其他相关信息，包括课程编号、课程名称、课程地点、课程教师、QQ群号、课程时间、考试安排、作业情况等。

(1.2) 课程输入要求

满足要求，学生可以通过前端的课程表或课程界面来查询课程的详细信息。

(1.3) 课程查询要求

满足要求，用户输入方式为直接在前端界面进行对应操作，可以通过不同关键字进行排序。

(1.4) 课程资料要求

满足要求，支持课程资料以及作业的上传下载以及查重和版本管理，支持按照名称以及上传时间进行排序，支持资料的查询和选定。版。本下载。

11.1.2 课外信息管理和查询

(2.1) 课外活动数量要求

满足要求，课外活动有 22 个，包括篮球赛、班会、自习等。

(2.2) 课外活动查询与排序要求

满足要求，用户可以在前端日程表界面查询课外活动信息，可以通过活动类型进行查询，可以对查询结果进行多关键字排序。

(2.3) 课外活动提醒要求

满足要求，可以设置活动闹钟，并选择重复情况。

(2.4) 课外活动冲突检测要求

满足要求，可以进行冲突检测。

11.1.3 课程导航

(3.1) 校园内建筑物数量要求

满足要求，包含沙河和西土城两个校区。沙河校区有38个建筑物，边数200条；西土城校区有64个建筑物，边数300条。

(3.2) 导航基本要求

完成，我们可以允许用户在导航主界面自行设置远地点和目的地点，完成对应策略的导航，并输出导航路线。根据课程时间的导航接口位于日程表界面，需要在日程表界面选择时间，即可自动跳转到导航主界面并设置目的地；根据课程名称的导航接口位于课程界面，需要在课程界面点击导航按钮，即可自动跳转到导航主界面并设置目的地。

(3.3) 导航策略要求

满足要求，可以根据最短距离、最短时间（含拥挤度计算）、交通工具最短时间规划导航路线，可以输出多条最短路。

(3.4) 导航扩展要求

满足要求，可以实现跨校区的路线规划。

11.1.4 模拟系统时间

(4.1) 模拟时间推演要求

满足要求，系统时间以小时为单位，可以模拟时间推移，并在人机交互时暂停。

11.1.5 日志

(5.1) 日志文件要求

满足要求，可以在前端随时查询日志信息，可以根据日志恢复现场。

11.1.6 选做

(6.1) 设计各种功能的图形界面

完成，我们使用 QT 设计并实现了完整的此系统的 UI 界面。

(6.2) 途经多个地点的最短距离路径

完成，我们采用了动态规划算法，可以实现计算途经多个地点的最短路径。

(6.3) 能够使用课表图形界面方式进行课程管理和查询

完成，我们设计了课程表界面，此外还有课程管理的 UI 界面，可以图形化地展示课程并允许用户进行修改、查询、排序。

(6.4) 能够对课程作业和资料进行版本管理

完成，我们实现了所有文件的历史版本维护管理。

11.2 改进意见

11.2.1 关于课程文件查找排序功能

实现时考虑到尽量做到接口和实现的统一，对于时间和名称的排序都是使用快速排序方法实现的。但是，快速排序的复杂度为 $O(n \log n)$ ，在存在平衡树的情况下，直接在平衡树上遍历或许查找效率更高，因此，如果放弃代码的工整性和易维护性，将快速排序用平衡树来替换，或许是一个提高算法效率的方法。如果进一步重构底层代码，优化底层接口设计，可以使得代码兼具高效和工整。但由于重构底层代码工作量较大，未来得及完成这项工作。

11.2.2 关于网络通信架构

由于前期阅读ppt时未考虑到做成网络应用的情况，因此我们并未开发网络版本。现阶段，我们的架构本身就是用户前端与后端分离的，两部分都封装良好，仅需将函数调用改为 socket 通信，即可将该系统部署为网络应用。后续可以进一步加上网络通信功能，使得用户使用更加便捷。由于时间原因，我们并没有增加与其他课程辅助软件的互通功能。后续也可以调用其他课程辅助软件的API，进一步实现作业提交至指定软件、平台的方法，使得系统和其他软件可以文件互通。

11.2.3 关于文件中文路径

由于 C++ 本身不支持中文路径，因此我们的文件上传和下载无法使用中文路径，如果遇到中文路径情况，程序会运行时错误然后退出。后续改进可以增添加入中文路径的方法。

12 用户使用说明

12.1 起步

本系统为线下课程辅助系统，为学生和学校管理员（教师）提供了便捷、简单的校园生活辅助服务，帮助师生完成课程信息管理、课外活动管理、日程表查看、闹钟设定、资料作业上传下载等日常校园活动和工作。此外，本系统还提供了时间模拟与日志文件功能，便于用户观察模拟时间推进下系统的变化，并通过阅读日志文件了解系统的工作状态。

此文档分主题的介绍了线下课程辅助系统的主操作界面、校园导航、日程表、课程管理、管理员与用户权限等功能，帮助使用者快速了解本软件的使用方法。

12.2 系统结构

本系统包括日程管理系统、校园导航系统、课程管理系统、时间模拟系统、日志文件系统五大模块，分别提供不同的服务功能。对于用户来说，本系统包括登录界面、学生主界面、管理员主界面、日程表子界面、校园导航子界面、课程管理子界面、闹钟子界面几大界面。

本文档会一一介绍这些界面的功能，并指引用户如何进行各种操作。

• 登录界面



• 学生主界面



• 管理员主界面

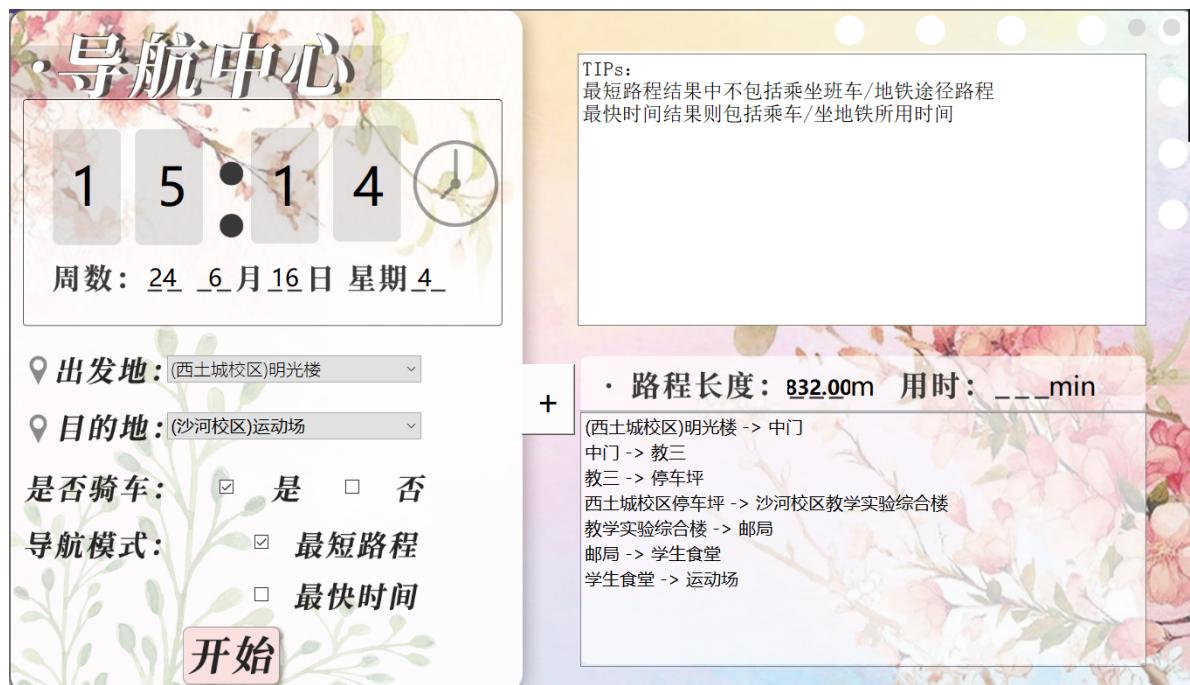


• 日程表子界面

The screenshot shows the schedule sub-interface. At the top right is a date and time indicator: "周数: 24_6月14日 星期2_". Below it is a search bar labeled "请输入查询活动的名称" and two radio buttons for "个人" (Individual) and "集体" (Group). The main area is a weekly calendar grid from Monday to Sunday. The grid shows various scheduled activities with specific details like room numbers and times. A pink floral pattern serves as the background for the grid. A large blue plus sign is located in the bottom right corner of the grid.

	Mon.	Tue.	Wed.	Thur.	Fri.	Sat.	Sun.
8:00	计算机网络 沙河校区 教学实验综合楼 S516 课程或考试	形式语言与自动机 沙河校区 教学实验综合楼 S510 课程或考试	计算机网络课程设计 沙河校区 教学实验综合楼 N210 课程或考试			高等数学 西土城校区 主楼 N110 课程或考试	
9:00							
10:00			数字逻辑课程设计 沙河校区 教学实验综合楼 N210 课程或考试	数据结构课程设计 沙河校区 教学实验综合楼 N214 课程或考试			
11:00	计算机组成原理 沙河校区 教学实验综合楼 S510 课程或考试				线性代数 西土城校区 主楼 N508 课程或考试		
12:00							+ (blue)
13:00							

• 校园导航子界面



• 课程管理子界面

课程管理

WEEK: 24 DAY: 15

请输入文件名/资料名

作业

资料

考试

通知

当前课程

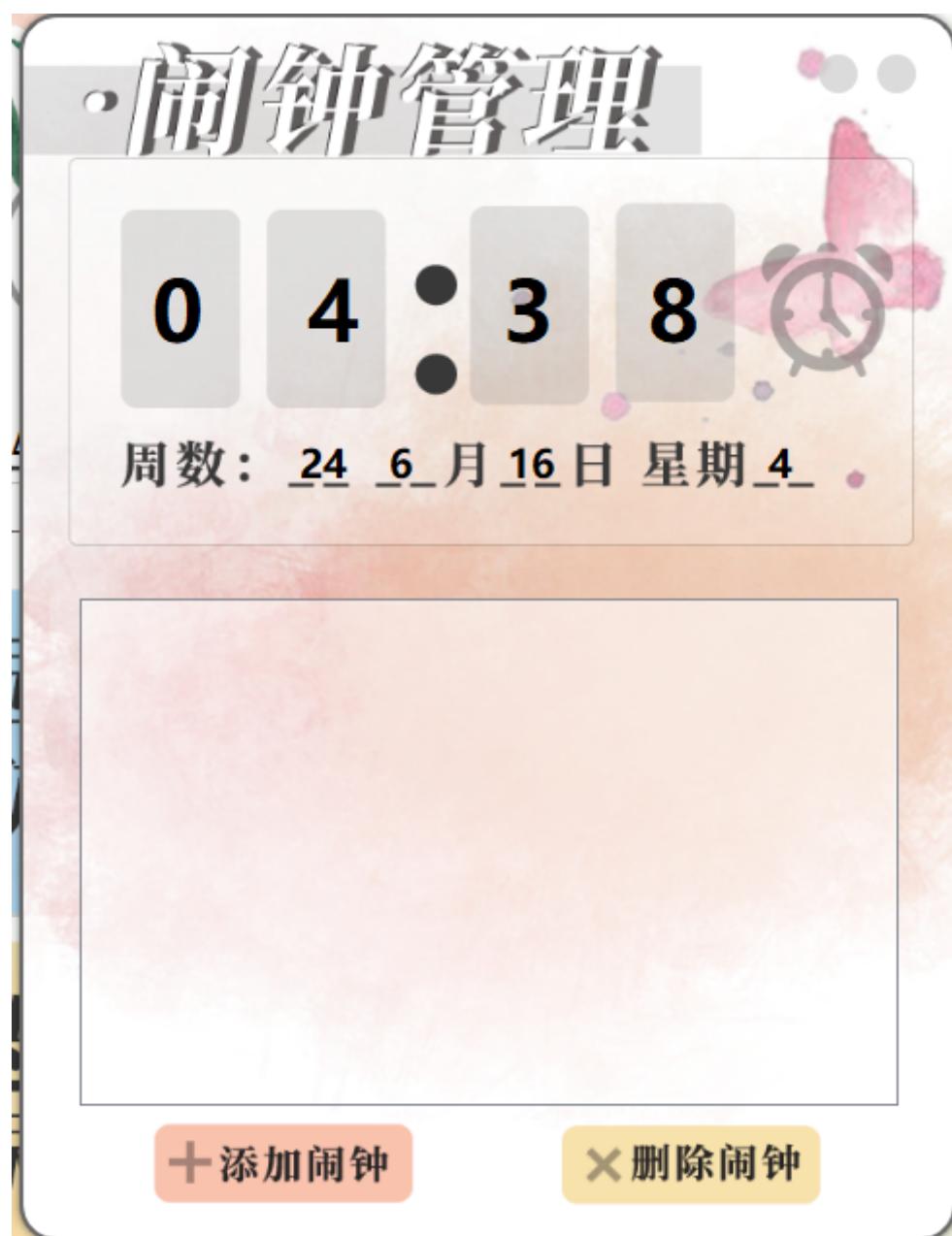
面向对象程序设计
上课时间: 13:00
下课时间: 15:00
上课地点: 教学实验综合楼 N210
QQ群为: 00005753200
老师为: 9老师

名称	所属课程	上传时间	原文件名 (最新版本)
作业一	计算机网络	第23周 星期7 20:00	QtV2.pdf
作业三	计算机网络	第23周 星期7 0:00	training model ...
作业二	形式语言与自动机	第23周 星期7 4:00	FlashOS.h

○ 按时间排序 ● 按名称排序

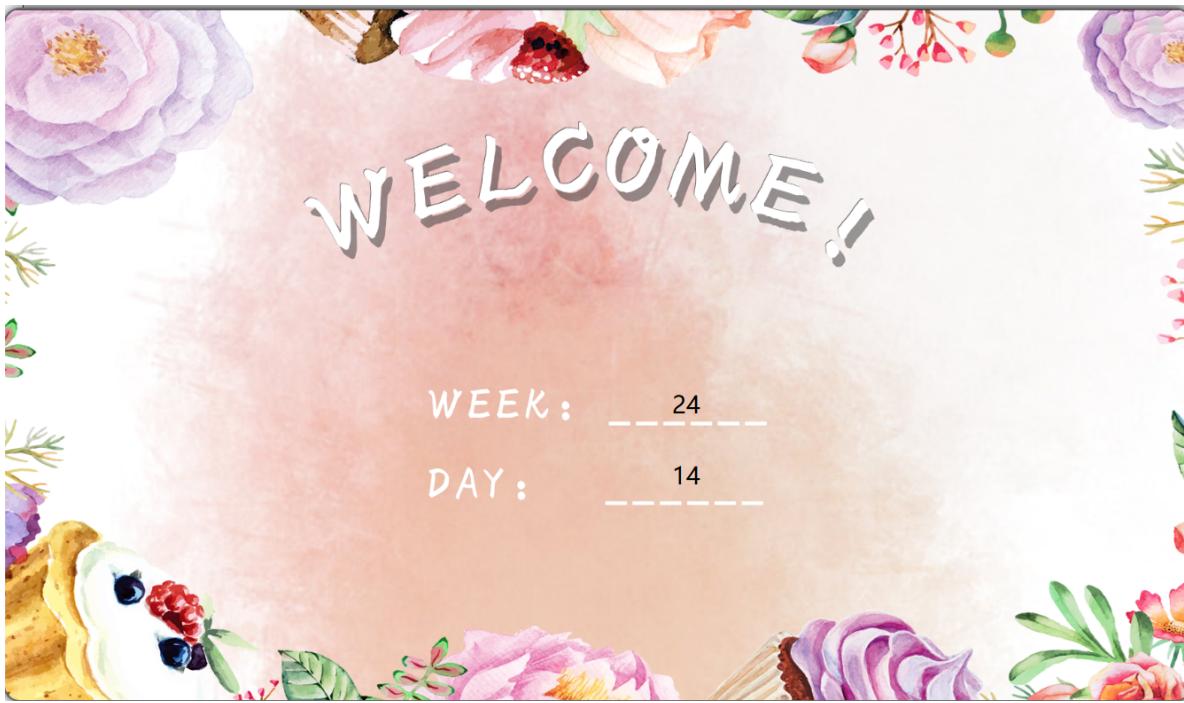
添加课程 提交作业 上传资料

- 闹钟子界面



12.3 登录界面

软件运行后，会先出现开始界面。



在程序运行的开始会显示如上图所示的欢迎界面，再点击后会有简单的过场动画切换到登录界面：



在用户输入账户以及密码后则会切换至主界面。

12.4 学生主界面

学生模式下，界面左上方会显示当前的模拟时间，在时间旁边有个时钟图标的按钮，再点击后会显示闹钟界面，左下方则是五个按钮，点击后会打开对应功能的页面。在右侧下方的通知公告区域会显示管理员发布的考试通知，作业通知或者是上课时间地点变更通知。管理员模式按钮则是在点击后切换为管理模式并更改界面图示以及按钮功能。



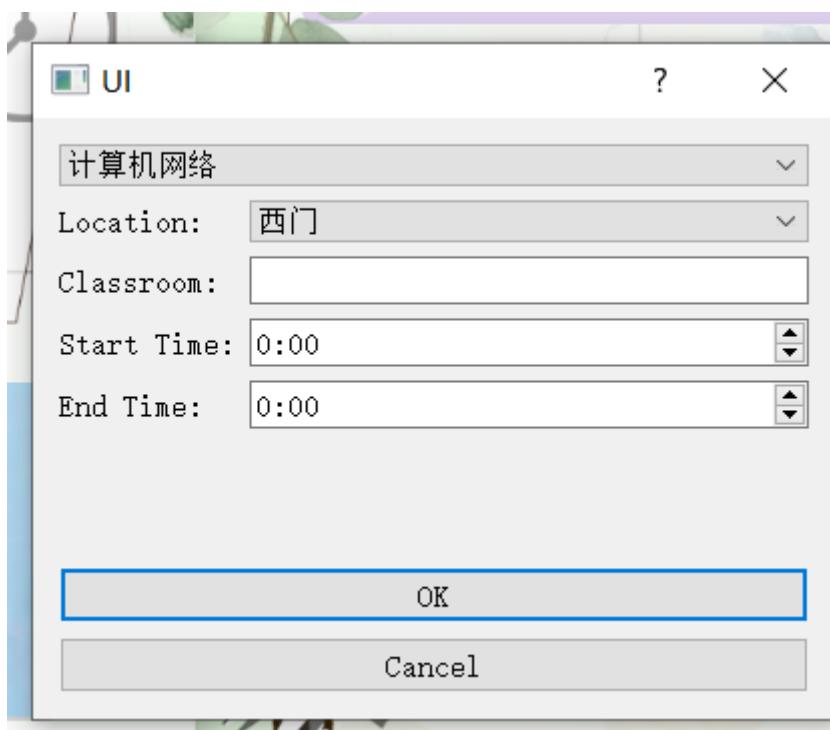
12.5 管理员主界面

在管理员模式下，主要变化的是右下角的按钮，分别用于实现上课时间地点的变更，发布新的作业以及考试时间发布以及显示本次运行时产生的纪录（该功能与用户模式下相同）。

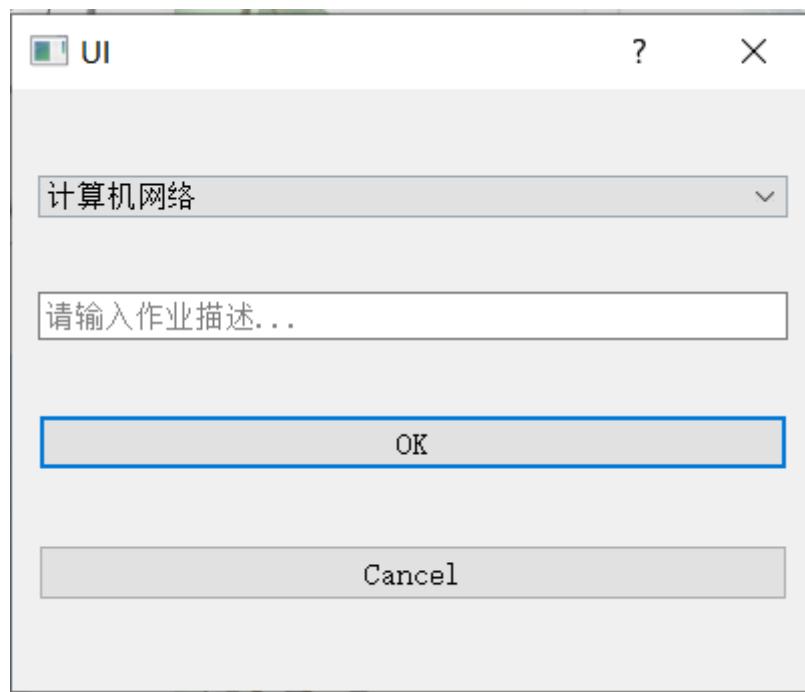


下面简单演示几个管理员界面的功能：

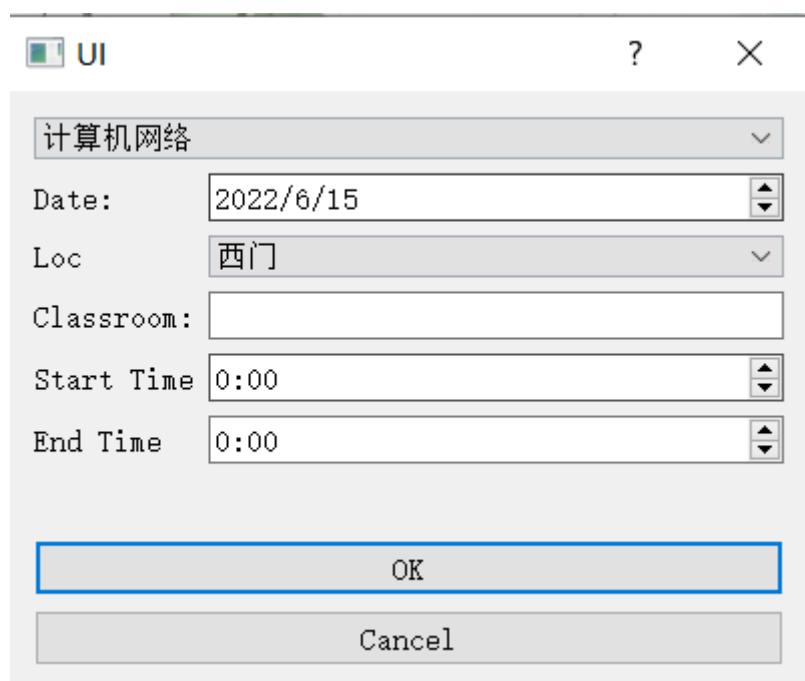
- 通知发布（上课时间地点的变更）



- 作业情况 (发布新作业)



- 考试发布

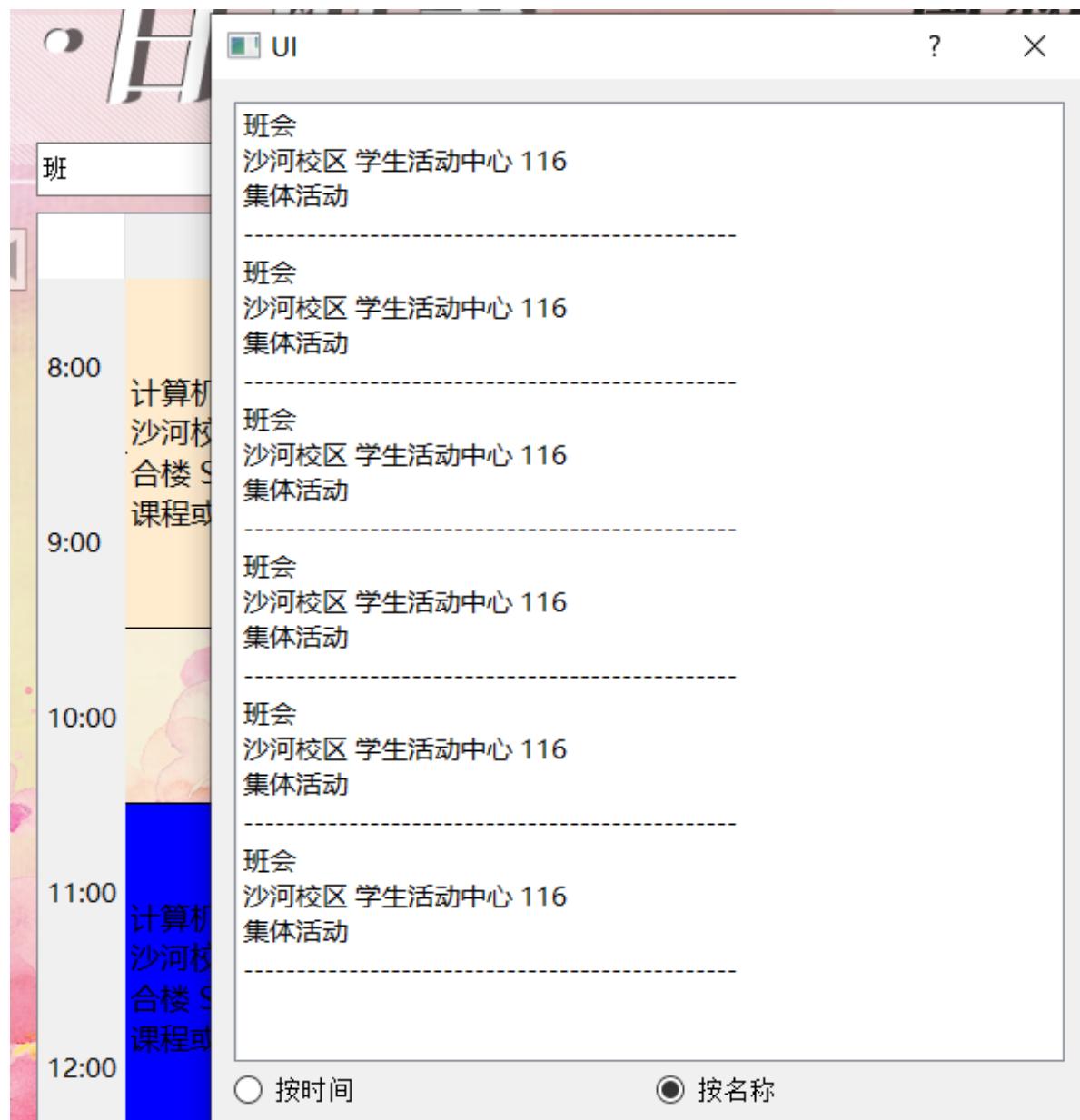


12.6 日程表子界面

日程表界面：



日程表界面则是按周展示该周的全部日程。所有的课程与活动会以色块的形式展示在日程表上。在界面的右上角活动查询部分，在输入活动名称后输入回车即可模糊搜索匹配上输入字符串的活动，并支持按照名称以及时间进行排序，演示如下图所示：



点击下方的选框，可以切换排序方式，可选排序方式有按名称排序和按时间排序。

类似的，界面右上角个人或集体的选框则是针对活动类型对活动进行查询，同样支持按时间或按名称排序。

篮球赛

沙河校区 运动场 篮球场

个人活动

篮球赛

西土城校区 篮球场 篮球场

个人活动

活动闹钟测试

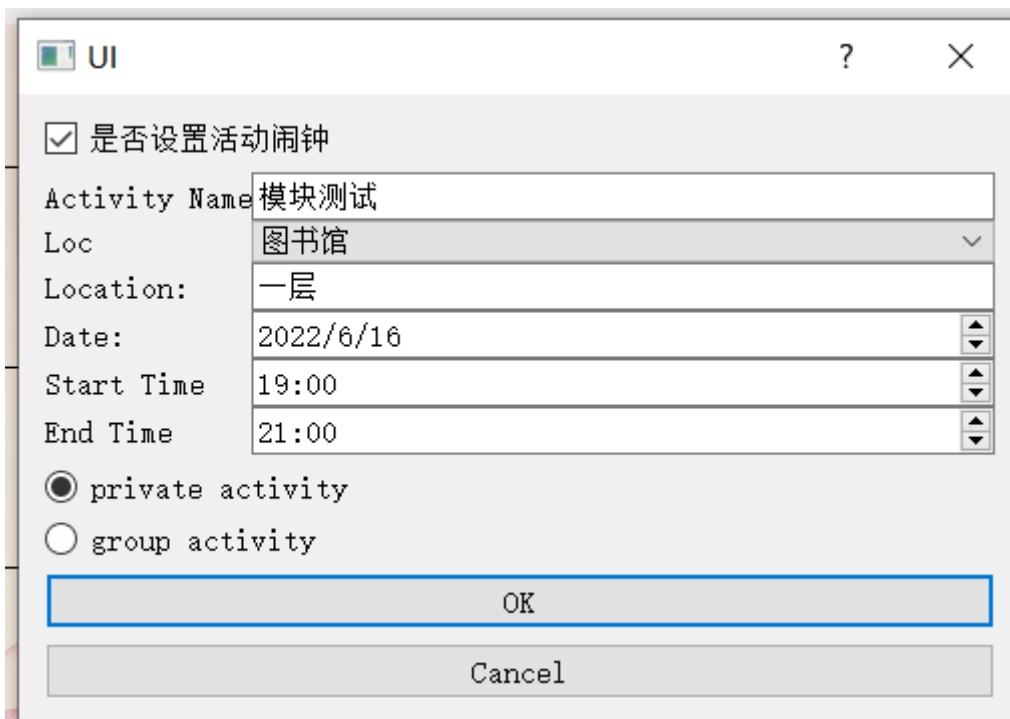
沙河校区 综合办公楼 203

个人活动

按时间

按名称

在界面右下角有一个加号按钮，用于添加活动，并能检测新活动同所有日程的时间冲突并给出提示。同时可以通过选择设置活动闹钟自动在活动添加成功时设置闹钟提醒：



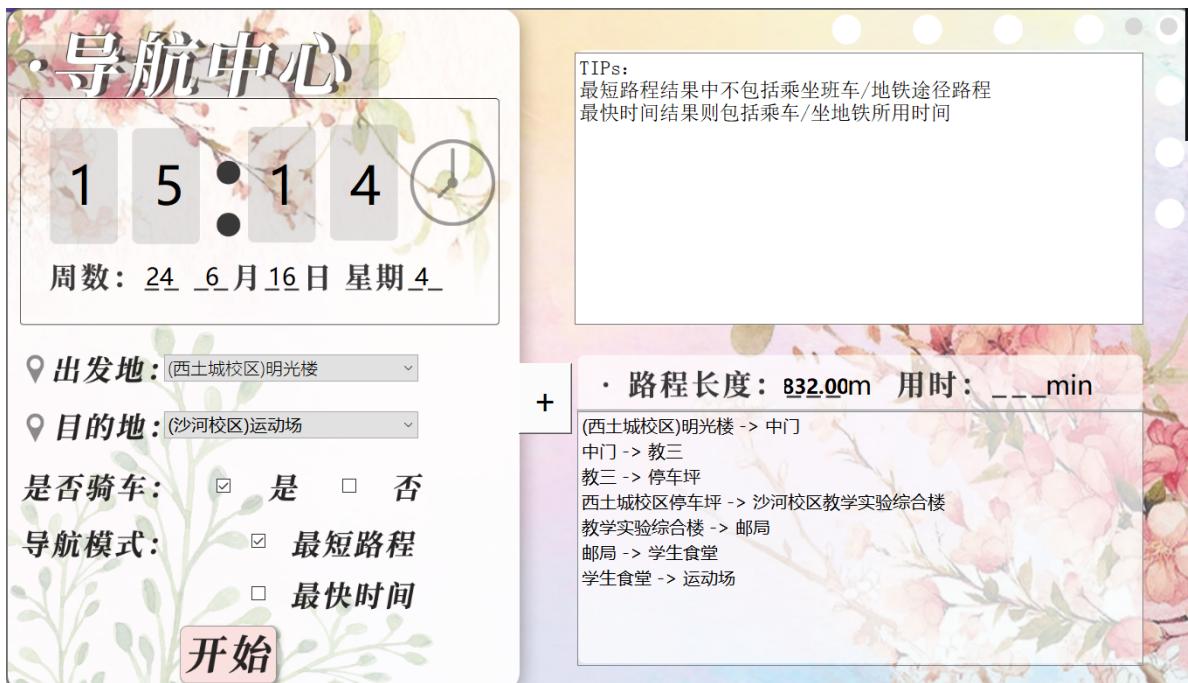
页面上方分别有两个箭头按钮用于切换周数显示非本周的日程，并且在此时界面上方的日期颜色将会转变为红色，如下图所示：



此外，双击日程表非空格子（即有日程的格子），系统将会自动获取双击日程的举办地点并跳转到导航页面，将目的地设置为当前选中日程的举办地点。

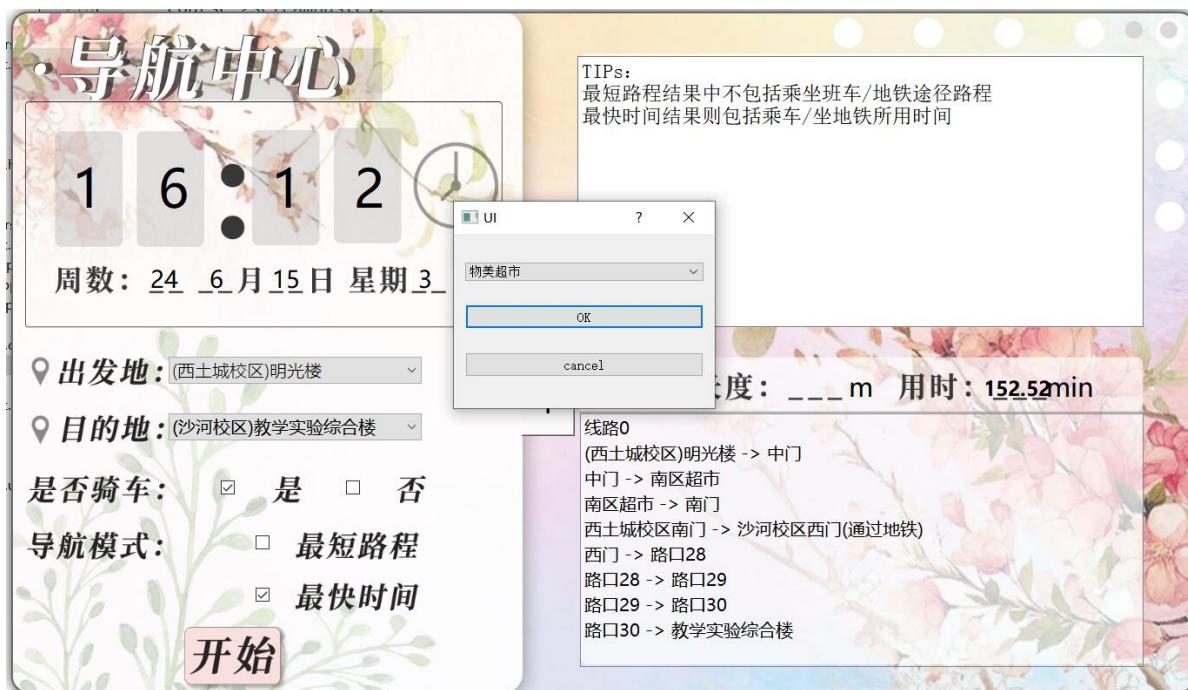
12.7 校园导航子界面

导航界面：



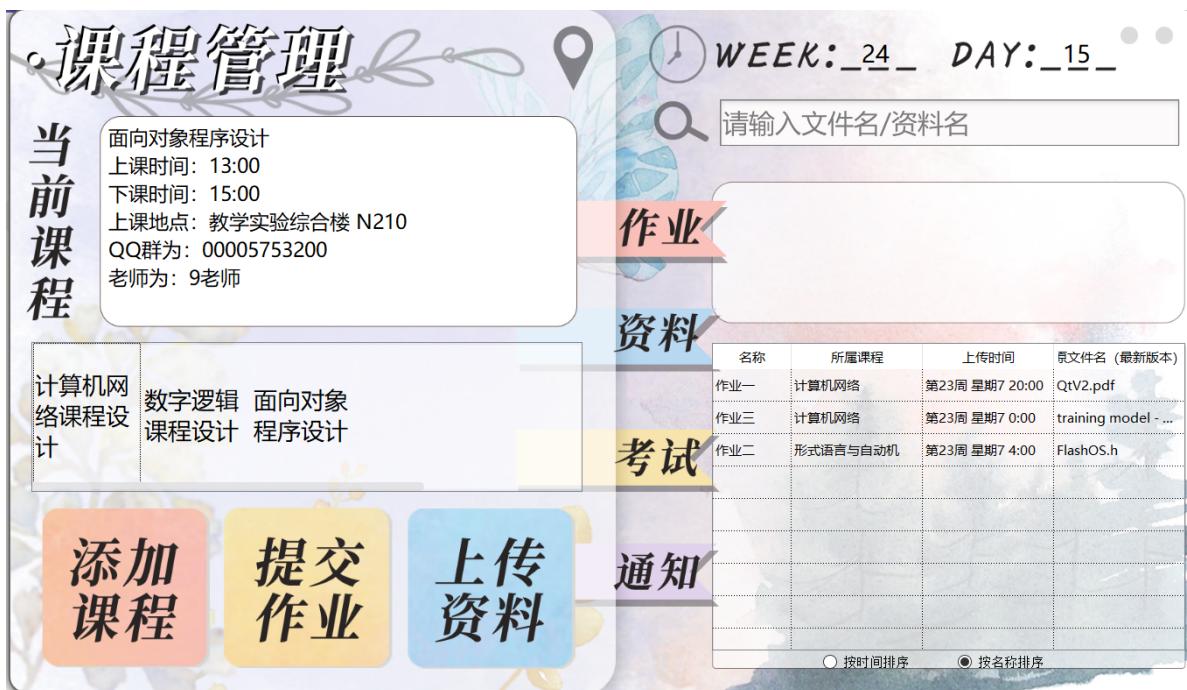
在导航界面中，用户可以在选择出发地、目的地，设置好途经点（点击页面中心偏左的加号）并选择好导航策略后点击开始按钮进行导航，导航结果将在右下角进行展示（如图所示）。

通过左侧的加号可以添加途经点，弹出的窗口如图：



12.8 课程管理子界面

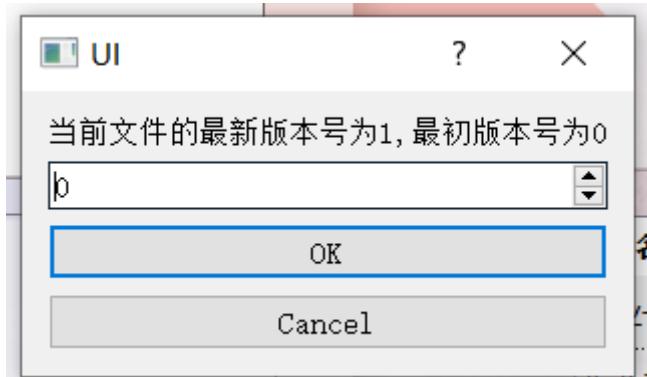
课程管理界面：



该界面的上半部分显示当前上课的详细信息，页面中部则是今天全天的课程（按时间排序），同时在双击课程名称后也可以自动获取上课地点并跳转到导航界面。左下方则是三个按钮，分别完成选课功能，提交作业以及上传功能。

在界面的中间，有四个按钮，点击后完成模式的切换。

上述截图中展示的是作业模式，右侧的搜索栏用于模糊搜索作业，并在下方的列表栏进行展示，下方的表格则是展示全部作业，并可通过底部两个按钮切换排序方式。在双击搜索结果或者表格中某项则会弹出窗口选择版本下载到指定地址。



点击“资料”后，跳转至资料模式。资料模式与作业模式类似，只是搜索以及展示的文件由作业变为课程资料：

课程管理

WEEK: _24_ DAY: _15_

当前课程

面向对象程序设计
上课时间: 13:00
下课时间: 15:00
上课地点: 教学实验综合楼 N210
QQ群为: 00005753200
老师为: 9老师

计算机网络课程设计 数字逻辑 面向对象课程设计 程序设计

添加课程 提交作业 上传资料

作业 资料 考试 通知

请输入文件名/资料名

名称	所属课程	上传时间	原文件名 (最新版本)
第一章资料	计算机网络	第23周 星期7 5:00	armcc_intr.h

○ 按时间排序 ○ 按名称排序

点击“考试”后，跳转至考试模式。考试模式下展示已经发布的考试信息：

课程管理

WEEK: _24_ DAY: _15_

当前课程

面向对象程序设计
上课时间: 13:00
下课时间: 15:00
上课地点: 教学实验综合楼 N210
QQ群为: 00005753200
老师为: 9老师

计算机网络课程设计 数字逻辑 面向对象课程设计 程序设计

添加课程 提交作业 上传资料

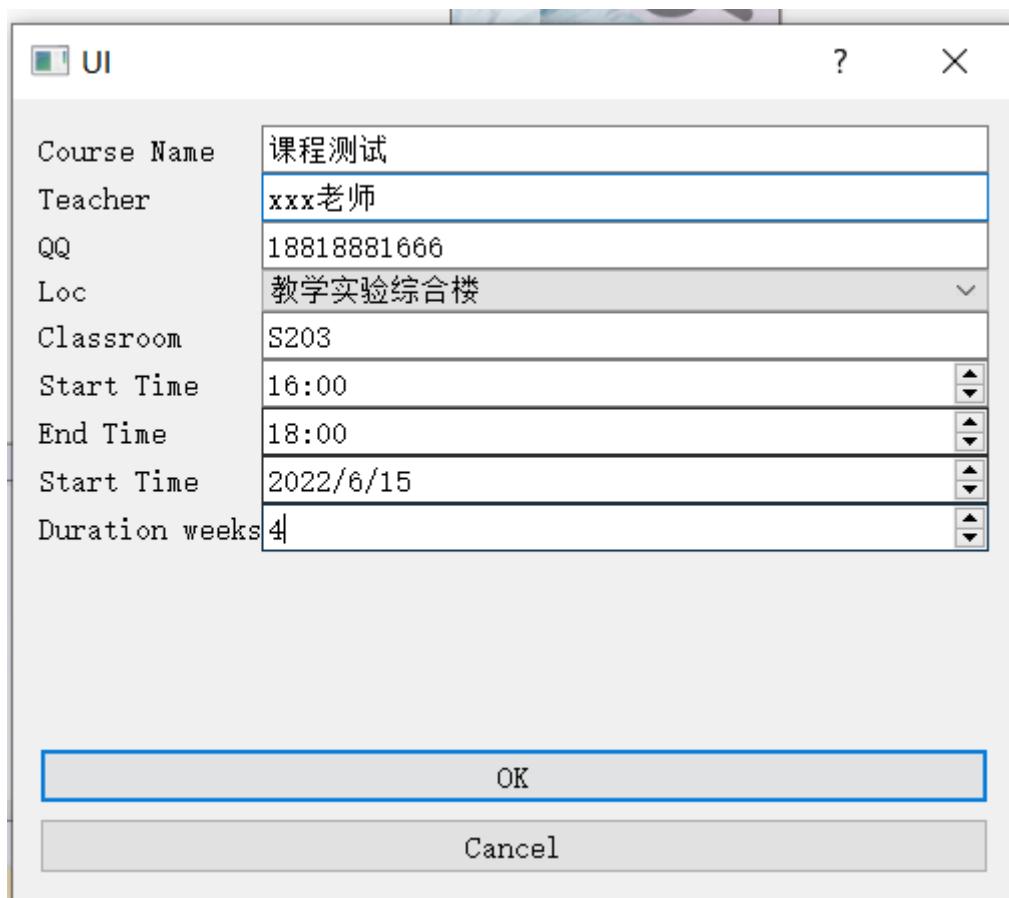
作业 资料 考试 通知

计算机网络考试时间为:
第28周 星期1 10:00-12:00
计算机组成原理考试时间为:
第28周 星期2 8:00-10:00

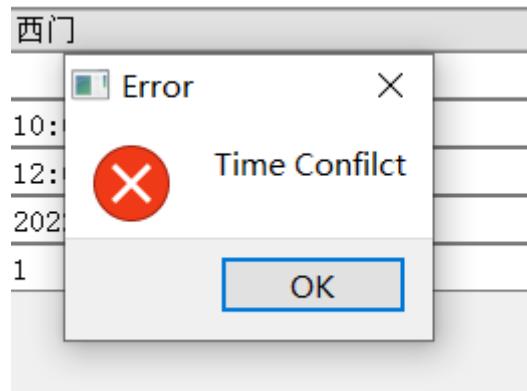
点击“通知”后，切换到通知模式。通知模式下展示当前已经交的作业以及未交作业的名录：



点击“添加课程”按钮，可以进行课程的添加。会弹出对应的对话框，设置相关信息后完成课程的添加：

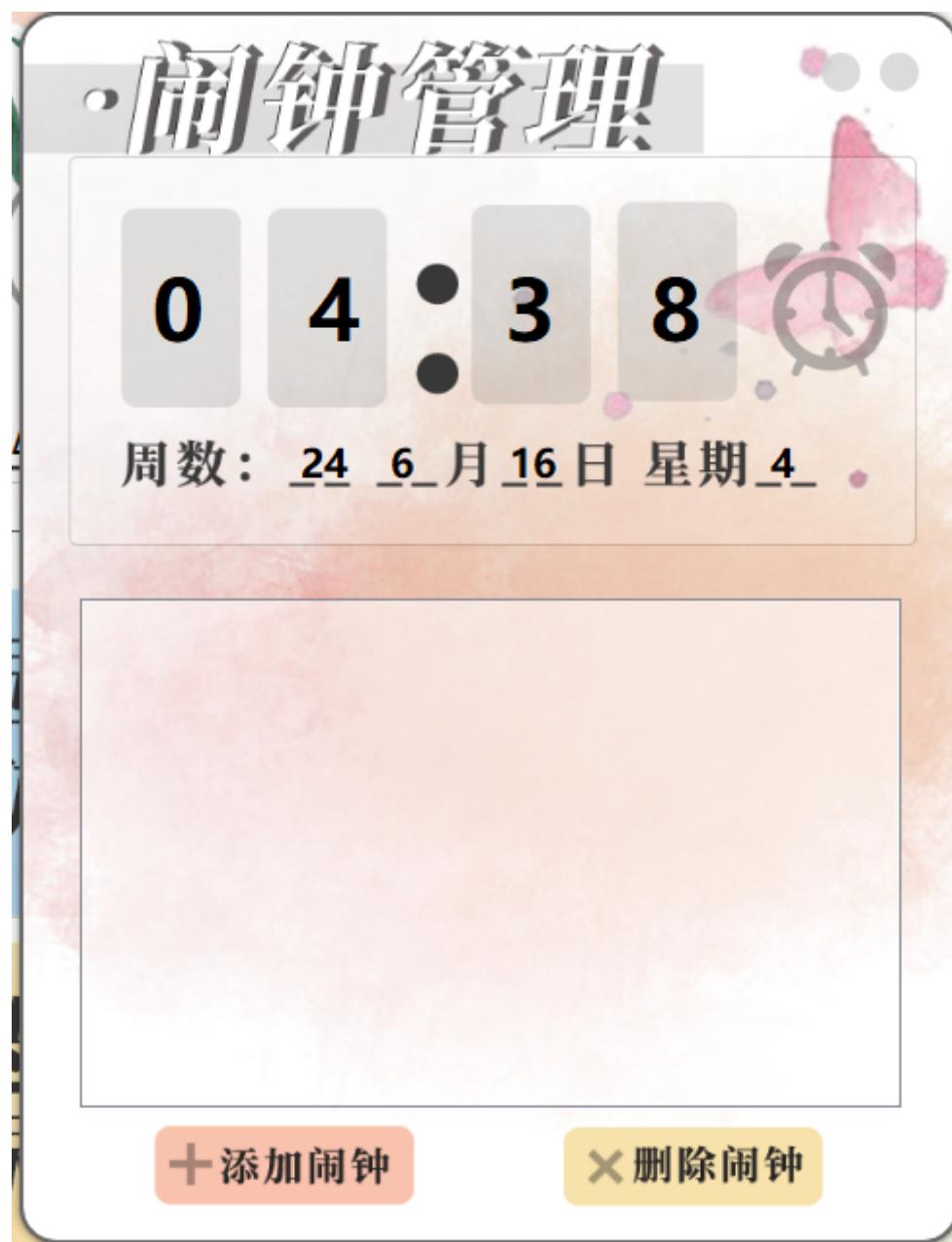


如果正添加的课程与已有课程冲突，会弹出警告：



12.9 闹钟子界面

闹钟界面：



用户只需点击添加闹钟并填写相关信息即可设新的闹钟，全部闹钟将会在界面下方列表显示，用户在选中某一闹钟后在点击删除闹钟按钮即可删除某闹钟。

若用户此前添加过活动闹钟，那么可以在此界面进行闹钟查看和管理。

闹钟管理

2 1 : 0 0



周数: 24 6月 15日 星期 3

模块测试日程 19:00:00 OnlyOnce

模块测试 19:00:00 OnlyOnce